2010-12

Design and implementation of an audit subsystem for a separation kernel

Toh, Boon Pin

Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/4971

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

DESIGN AND IMPLEMENTATION OF AN AUDIT SUBSYSTEM FOR A SEPARATION KERNEL

by

Boon Pin Toh

December 2010

Thesis Co-Advisors: Cynthia E. Irvine
Paul C. Clark

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** December 2010	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE Design And Implementation of an Audit Subsystem for a Separation Kernel			**5. FUNDING NUMBERS**
6. AUTHOR(S) Boon Pin Toh			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number ________________.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

A separation kernel can be used as the foundation of a high assurance system that enforces mandatory security policies. The contexts in which such separation kernels might be used include support for a distributed trusted path, high assurance routing, and for a multilevel secure mobile device that supports an extraordinary access partition for access to sensitive data during a crisis. Separation kernel requirements call for an audit subsystem that helps to enforce accountability policy by allowing administrators to detect unauthorized activities from the logs collected. The Least Privilege Separation Kernel (LPSK) being implemented for the Trusted Computing Exemplar (TCX) project did not have an audit subsystem.

This thesis describes the design and implementation of an audit subsystem for the LPSK. Requirements were gathered based on an existing specification and protection profile. A variable-length token-based audit log format was designed to allow flexibility in recording different types of events. Interfaces to other LPSK modules and non-LPSK modules were designed and a prototype was developed. Testing results show that the prototype supports the LPSK audit requirements. Hence, this work demonstrates the feasibility of implementing the LPSK audit subsystem based on the proposed design.

14. SUBJECT TERMS Separation Kernel, Audit, High Assurance System			**15. NUMBER OF PAGES** 133
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

DESIGN AND IMPLEMENTATION OF AN AUDIT SUBSYSTEM FOR A SEPARATION KERNEL

Boon Pin Toh
Civilian, Defence Science & Technology, Singapore
BSc, University of Tokyo, 2002

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 2010

Author: Boon Pin Toh

Approved by: Cynthia Irvine
Thesis Co-Advisor

Paul C. Clark
Thesis Co-Advisor

Peter J. Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

A separation kernel can be used as the foundation of a high assurance system that enforces mandatory security policies. The contexts in which such separation kernels might be used include support for a distributed trusted path, high assurance routing, and for a multilevel secure mobile device that supports an extraordinary access partition for access to sensitive data during a crisis. Separation kernel requirements call for an audit subsystem that helps to enforce accountability policy by allowing administrators to detect unauthorized activities from the logs collected. The Least Privilege Separation Kernel (LPSK) being implemented for the Trusted Computing Exemplar (TCX) project did not have an audit subsystem.

This thesis describes the design and implementation of an audit subsystem for the LPSK. Requirements were gathered based on an existing specification and protection profile. A variable-length token-based audit log format was designed to allow flexibility in recording different types of events. Interfaces to other LPSK modules and non-LPSK modules were designed and a prototype was developed. Testing results show that the prototype supports the LPSK audit requirements. Hence, this work demonstrates the feasibility of implementing the LPSK audit subsystem based on the proposed design.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

BSM	Basic Security Module
CC	Common Criteria
CISR	Center for Information Systems Security Studies and Research
EAL	Evaluation Assurance Level
LPSK	Least Privilege Separation Kernel
NIST	National Institute of Standards and Technology
SAK	Secure Attention Key
SKPP	Separation Kernel Protection Profile
TCB	Trusted Computing Base
TCX	Trusted Computing Exemplar
XML	Extensible Markup Language

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank my thesis advisors, professors Cynthia Irvine and Paul Clark, for the valuable advice and guidance they have provided throughout the thesis writing process. I would also like to thank David Shifflett for his technical expertise and assistance.

I would also like to thank my sponsor, the Defence Science & Technology Agency (DSTA), for supporting my study at Naval Postgraduate School.

Finally, I would like to thank my family and friends for their support and encouragement.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. MOTIVATION

To reduce software, hardware and management costs, organizations might choose to use a single system to handle multiple types of information, which could include information of different sensitivity levels. In such as system, the assurance that the security policy is properly enforced to protect against unauthorized information flow is of utmost importance.

In certain scenarios, timely availability of information could be critical in certain circumstances, such as during emergencies. For example, the first responders might need access to sensitive information not normally available to them. A high assurance multilevel secure device such as the E-device [1] proposed by the Naval Postgraduate School, supports this kind of operation by allowing the users to switch a platform to an emergency mode for the duration of the crisis, which gives them the necessary access.

The Naval Postgraduate School Trusted Computing Exemplar (TCX) project [2] is developing a high assurance platform that that could provide solutions to the use cases described above. The TCX objectives include a high assurance reference implementation that includes a Least Privilege Separation Kernel (LPSK). The LPSK isolates the resources into different partitions and has granular control over the configuration of information flow between the partitions.

The LPSK is built to comply with the U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness (SKPP) [3]. The SKPP is a requirements document that contains the security objectives, functional requirements and assurance requirements for a separation kernel. The SKPP mandates that any separation kernel, including the LPSK, that seeks certification against it must minimally fulfill the audit requirements stated in the protection profile. Separation kernels might also include additional audit requirements depending on their specific implementations.

National Institute of Standards and Technology (NIST) Special Publication 800-53 [4] describes the recommended security controls for federal information systems and

organizations. It mandates the implementation of audit and accountability that can uniquely trace actions back to users. The proposed E-device, for example, must provide a mechanism to determine whether the first responders are gaining access to data beyond what is needed to accomplish their missions. An effective audit subsystem could facilitate an after action review by rebuilding the entire chain of events that occurred during the crisis.

B. PURPOSE OF STUDY

The purpose of this thesis was to design and implement a prototype of an audit subsystem for the LPSK. The requirements were gathered from the LPSK functional specifications [5] and the SKPP. The objective of the design was to seek answers to the following questions:

- What factors must be considered when designing an audit subsystem for a separation kernel?
- What information should the audit records contain and in what format?
- What interfaces are needed for the audit subsystem to interact with other components?
- Can an audit subsystem constructed as part of the LPSK prototype meet the requirements of the LPSK functional specifications and SKPP?

An audit subsystem prototype of was developed to demonstrate the feasibility of the design. A test plan was devised and the prototype was tested to ensure it behaves according to specifications.

C. ORGANIZATION OF PAPER

The thesis is organized into six chapters. Each chapter is systematically organized to provide an in-depth discussion of the different aspects of the thesis.

- Chapter I introduces this thesis. The motivation and purpose of study were discussed.

- Chapter II provides the background information on separation kernels, the Principal of Least Privilege, the Common Criteria and specifically the SKPP, the TCX project, the purpose of auditing, log management architectures and audit record standards.
- Chapter III describes the requirements for the LPSK audit subsystem. It includes discussions of audit record event selection, automatic response, data generation and data review. It also provides a list of events to be audited.
- Chapter IV describes the design and implementation of the LPSK audit subsystem. It starts with a presentation of a high level design to introduce the various components involved in the audit subsystem. This is followed by detailed discussions of audit record formats and how the components interface with one another. This chapter ends with a description of the implementation of the audit subsystem prototype.
- Chapter V describes how the LPSK audit subsystem prototype was tested according to developmental and acceptance testing plans.
- Chapter VI provides a summary of the work that has been done for this thesis and a discussion of the challenges faced. Related work is also discussed to compare the different approaches used in similar projects. This is followed by a conclusion of the thesis and suggestions for future work.

This chapter introduced the thesis by describing the motivation and purpose of study, and gave an overview of the organization of the paper. Focus now changes to background material needed to appreciate the work that follows.

THIS PAGE INTENTIONALLY LEFT BLANK

II. BACKGROUND

This chapter provides background information regarding separation kernels and the necessary audit mechanisms to support them. The concept of a Least Privilege Separation Kernel will be introduced, followed by a discussion of the purposes and functions of an audit mechanism.

A. LEAST PRIVILEGE SEPARATION KERNEL

This section discusses the importance of a high assurance system and the benefits of constructing such a system using a Least Privilege Separation Kernel.

1. Separation Kernel

While Commercial off-the-Shelf (COTS) systems may be sufficient in handling the security requirements for general tasks in the private and public sectors, they are not designed to protect highly sensitive information. COTS systems are usually not easily verifiable, nor are they able to enforce multilevel security policies and address the problem of subversion. This is where high assurance trusted computing systems are needed.

One way to design a high assurance system is to put a *security kernel* at its core. A security kernel consists of core security components that will mediate all data flow and accesses to resources. It is made up of hardware and software mechanisms that fall within the Trusted Computing Base (TCB), which is the totality of all protection mechanism responsible for enforcing a security policy. In a security kernel, an internal security label is bound to each exported resource and accesses to the resources are mediated according to predefined security policies based on these labels. Efforts are usually made to keep the security kernel small enough to be formally verifiable.

Despite these efforts, some argue that security kernels are too large. Another architecture is a *separation kernel* proposed by Rushby [6]. The idea behind a separation kernel is to provide a single system that emulates a number of distributed systems in which the components are physically separated into different isolated blocks. Information

flow is described at the block level. The kernel can export the resources to separate blocks such that the activities in one block will not be visible to other blocks. An exception might be when information flow between the two blocks is explicitly allowed in the configuration. Isolation of blocks is usually accomplished by virtualization of shared resources and implementation of security mechanism controls to enforce policies. Security is achieved through this isolation and through the mediation of trusted functions.

2. Principle of Least Privileged on Separation Kernel

In a separation kernel, if information flow is explicitly allowed between two blocks, all subjects in one block can see all the activities in the other block, even if the original intention is to only allow a small subset of the subjects to access a small subset of resources in the other block. The problem of describing information flow at the block level is that the policy configuration is not granular enough to handle individual subject-to-resource controls. This limitation means that the information flow configuration in a separation kernel is more likely to violate the Principle of Least Privilege [7]. The Principle of Least Privilege states that every subject must be able to access only such resources that are necessary for its legitimate purpose and nothing more than that. It is one of the major design principles that all secure systems should adhere to.

The Center for Information Systems Security Studies and Research (CISR) at the Naval Postgraduate School (NPS) is designing and building a separation kernel that will support the Principle of Least Privilege; it is referred to as the Least Privilege Separation Kernel (LPSK) [2]. The NPS LPSK extends the concept of separation kernels and adds mechanisms to allow more granular control. In addition to a policy that describes data flow between blocks, the LPSK mediates access based on another overriding subject-resource flow matrix. A subject is only allowed access to a resource if both the inter-block data flow policy and the subject-resource flow is allowed. The Principle of Least Privilege is fulfilled by granting the least set of privilege to resources in the LPSK.

3. Common Criteria and Protection Profiles

A high assurance separation kernel must demonstrably meet its security objectives through a thorough and comprehensive evaluation process. The Common Criteria (CC)

[8] provides a framework in which systems can be evaluated to determine whether they have met a required level of security functionality and assurance. The CC has been jointly developed and recognized as a security standard by many countries. When a system is being developed with the intent of meeting CC criteria, the developers target a specific Evaluation Assurance Level (EAL). The EAL ranges from EAL1, which is the lowest level of assurance in the CC framework, to EAL7.

The CC paradigm uses protection profiles as high level requirements documents. A protection profile contains the security objectives, functional requirements and assurance requirements for a particular category of system. The target system will be evaluated using the requirements stated in the protection profile. The U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness (SKPP) [3] contains such requirements for evaluating highly trustworthy separation kernels.

3. Trusted Computing Exemplar (TCX) Project

Even though the benefits of a high assurance Trusted Computing system are obvious, there has been very little work done on such systems in recent years. The Trusted Computing Exemplar (TCX) project [3] seeks to fill this gap by providing a worked example of a highly trusted computing system. The four main activities of the TCX project are:

- Creation of a prototype framework for rapid high assurance system development
- Development of a reference implementation trusted computing component
- Evaluation of the component for high assurance
- Open dissemination of deliverables related to the first three activities

A LPSK is being developed as part of the development of a reference implementation for the TCX project. This paper attempts to design and implement an audit subsystem based on the foundation of the TCX's LPSK.

B. AUDIT OVERVIEW

Audit is an integral part of any secure system. It generally refers to the mechanisms and process of recording, examining and reviewing of security-related operations to support organizational requirements for accountability. A document produced by the United States National Institute of Standards and Technology (NIST) that describes minimum security requirements for information systems [9] dictates that organizations must “create, protect, and retain information system audit records to the extent needed to enable the monitoring, analysis, investigation, and reporting of unlawful, unauthorized, or inappropriate information system activity”. It also requires organizations to “ensure that the actions of individual information system users can be uniquely traced to those users so they can be held accountable for their actions.” The purpose of audit actually extends far beyond ensuring accountability. This section summarizes some of the objectives of a good audit subsystem

1. Purpose of Audit

A good logging mechanism will facilitate the review of access patterns to individual objects [10]. An audit system must be able to facilitate the discovery of attempts by intruders to bypass the protection mechanisms, such as failed login attempts. This can be accomplished by regular inspections of the audit log by a security officer. The audit system can also act as a building block for other security components such as an *intrusion detection system*, which performs near real-time automated analysis of audit information to detect malicious attacks.

To protect against insider threats, the audit mechanism must also allow for the discovery of usages patterns that have violated or could be leading to a violation of an organization’s security policies [10]. It can also be used to monitor attempts to exploit covert channels. To provide these services, the audit mechanism must be able to track all users’ operations and the privileges they are assuming.

A good audit system can also act as a deterrent to potential attackers. Such individuals are less likely to carry out any malicious acts on the system if they know that

all their activities will be detected and recorded by the audit system. To the system owners, a good audit system also acts as a form of assurance that potential malicious activity will be discovered.

Besides contributing to security, a good audit mechanism also makes it easier for system developers and operators to troubleshoot the system in the event of system malfunction [11]. An administrator can review the audit logs to piece together information that can help him reconstruct the sequence of events, to identify what went wrong, and then potentially what needs to be corrected

2. Log Management Architecture

To be able to accomplish the goals of the audit subsystem, an effective way of generating, collecting and reviewing logs must be in place. A log management architecture addresses these issues by looking at ways to organize the various components to process and store audit information. A typical log management architecture usually consists of the following 3 tiers [12]:

- 1st Tier: Log Generation. Audit services run in the individual host/device to generate audit records. Audit mechanisms are usually part of the TCB of the system to ensure that they are tamper-proof, always invoked and verifiable. The events to be audited may be configurable to allow granular control over the amount of log information to be generated. There is a tradeoff decision that needs to be made when configuring audit, because a large number of generated audit records will provide more analytical data, but it will also require more CPU time and storage space, and may make it possible to hide malicious activity within an overwhelming amount of non-malicious activity. The audit services will make the log data available to log servers in the second tier through a protected network connection or other secure means.
- 2nd Tier: Log Analysis and Storage. This tier consists of log servers capable of collecting and storing log data from multiple hosts. Log data can be stored on the log servers themselves or on separate database servers.

- <u>3rd Tier: Log Monitoring.</u> This tier consists of consoles and tools that allow operators to monitor and review log data Tools can range from simple applications that allow operators to search and display audit records to IDS applications that perform real time monitoring of events.

The focus of this paper is to design the first tier log generation mechanism of the LPSK and to make provisions for the log data to be stored and retrieved for review in the second and third tiers of a log management architecture.

3. Audit Records Standard

An auditing system records important system events, where the data associated with each event is saved in the form of some kind of record. Even though most modern systems implement some form of audit, the industry as a whole lacks standards on the format of the audit records. A review of the formats used by the major operating systems today shows that almost all systems use their own proprietary audit record formats. This incompatibility often results in difficulties in log management, especially when events from different systems need to be combined [12]. This paper will give a more thorough discussion of the pros and cons of the different types of audit record format in Chapter 4.

C. SUMMARY

This chapter has provided background on the basic principles of the LPSK and how it contributes to the construction of high assurance systems. It has also given an overview of the purpose for the architecture of audit mechanisms for such a system.

III. REQUIREMENTS

The TCX LPSK functional specification [5] includes a list of requirements for its audit subsystem. As the LPSK is designed to be compliant with the SKPP [3], its audit subsystem must also fulfill the requirements of the SKPP. Below is a list of items that are mentioned in the TCX LPSK functional specifications and SKPP:

- Security audit event selection,
- Security audit automatic response,
- Security audit data generation, and
- Security audit review

Each of the above will be described in greater detail below.

A. SECURITY AUDIT EVENT SELECTION

Logging too little information is definitely not desirable, but logging too much may also be a problem. Logging will increase system overhead in terms of both storage space and processor time. This may result in a reduction in performance and significant reduction of storage space available for other processes in a system having tight resource constraints. Having too much audit log information may also increase the time needed for the operators to review the audit data. Thus, there is a need for the audit subsystem to offer the flexibility for administrators to specify the level of logging to be done on the system. This should be determined based on the operational requirements.

In the LPSK, the granularity of auditable events is defined using a configuration vector. A configuration vector is read by the LPSK during the kernel initialization phase and contains a set of information that describes the initial secure state of the LPSK platform and how the LPSK shall behave during the run-time. It also contains configurable options for the audit subsystem.

The SKPP requires that the separation kernel be able to include or exclude events from the runtime audited events based on the following attributes:

- Resource identity,
- Subject identity,
- Event type,
- Success of auditable security events, and
- Failure of auditable security events

Table 1 shows the optional auditable events that can be switched on or off based on the different choices in the configuration vector.

Table 1. Auditable Events via a Configuration Vector

Auditable Events	**Attributes**
When a signal is sent by a particular subject (success, failure, or both)	Subject identity
When a signal is received by a particular subject	Subject identity
When a software interrupt is invoked by a particular subject	Subject identity
When a device read is requested by a particular subject (success, failure, or both)	Subject identity
When a device write is requested by a particular subject (success, failure, or both)	Subject identity
When a device configuration is requested by a particular subject (success, failure, or both)	Subject identity
When the read of an eventcount is requested by a particular subject (success, failure, or both)	Subject identity
When the advance of an eventcount is requested by a particular subject (success, failure, or both)	Subject identity
When an await on an eventcount is requested by a particular subject (success, failure, or both)	Subject identity
When a process awakes from an await on an eventcount	Subject identity
When the ticket of a sequencer is requested by a particular subject (success, failure, or both)	Subject identity

When a read operation of a particular device is requested (success, failure, or both)	Device identity
When a write operation of a particular device is requested (success, failure, or both)	Device identity
When a configuration operation for a particular device is requested (success, failure, or both)	Device identity
When a ticket of a sequencer is requested (success, failure, or both).	Sequencer Event
When an advance of an eventcount is requested (success, failure, or both)	Eventcount Event
When a read of an eventcount is requested (success, failure, or both)	Eventcount Event
When an await on an eventcount is requested (success, failure, or both)	Eventcount Event
When a wakeup on an eventcount occurs.	Eventcount Event
When a particular segment is swapped in (success, failure, or both)	Memory segment Event
When a particular segment is flushed (success, failure, or both)	Memory segment Event
When a particular segment is swapped out (success, failure, or both)	Memory segment Event
When an mseg is created	Memory segment Event

Other configurable audit attributes in the configuration vector include:

- Enabling or disabling of audit
- Size of the audit buffer
- Action when audit buffer is full
    - Overwrite oldest record,
    - Halt the system, or
    - Shutdown the system

B. SECURITY AUDIT AUTOMATIC RESPONSE

The SKPP requires that the LPSK run a suite of self tests to verify both the hardware and software components of the kernel during start-up, periodically during normal operation, and during recovery. The audit subsystem shall record each of the failures and if required, the actions taken by the LPSK to recover from the failure. The audit subsystem may also be required to perform special actions such as halting the system upon detection of a critical failure during both the LPSK initialization and run-time phases. Table 2 shows a list of actions to be performed by the audit subsystem.

Table 2. Special Actions Taken by Audit Subsystem

Events	**Actions to be taken by audit subsystem**
Any audited event that causes the LPSK to halt the system	Display an informative message on the screen prior to the halt
Size of the audit buffer is specified outside the valid range	Display an informative message on the screen and halt the system
Failure of LPSK self-test	Record actions taken by the LPSK to try to correct the failure
Unsuccessful binding of security attributes to individual partitions	Display an informative message on the screen and halt the system
Attempt to recover the LPSK to a secure state	Record actions taken by the LPSK to try to recover (or halt the system)
Detection of invalid value or set of values in binary configuration vector during LPSK initialization	Halt the system
Inability of LPSK to return to a secure state after failure of a security function	Halt the system

C. SECURITY AUDIT DATA GENERATION

Based on the SKPP and LPSK requirements, a list of events has been identified as auditable. The audit subsystem shall be able to generate an audit record for each auditable event. In addition to the optional auditable events mentioned in section A, the following is a list of mandatory events that must be audited if audit is enabled in the configuration vector.

- Values of configuration vector
- Unsuccessful binding of security attributes to individual partitions, subjects, and non-subject exported resources
- The assignment of a default value to the configuration data during LPSK initialization
- The detection during LPSK initialization of an invalid value or set of values in a binary configuration vector
- The successful completion of LPSK initialization
- Successful start-up and shutdown of the LPSK audit mechanism by the LPSK Initializer
- Actions taken because of a failure of an LPSK self-test
- All requests for a configuration change
- The success of each startup of the LPSK
- A failure of an LPSK self test
- Any detected loss of secure state.
- Action taken to attempt to recover the LPSK to a secure state
- The inability of the LPSK to return to a secure state after failure of a security function.
- Changes to the LPSK time source

- Detection of a SAK invocation.
- The shutdown, power down or halt of a platform.
- Detection of duplicate MAC addresses

D. SECURITY AUDIT REVIEW

Audit records may contain sensitive information about the system. Thus, the audit services shall ensure that the records are only exported to authorized subjects. The audit subsystem shall store the audit records in an internal audit buffer and provide an external interface so that authorized subjects may obtain the buffered audit records. The LPSK functional specification requires the audit subsystem to provide call interfaces to retrieve the following information:

- Size of the audit buffer,
- Oldest buffered audit record, and
- Audit statistics
    - Number of audit records overwritten
    - Number of audit records generated

Audit records are not useful if they cannot be reviewed in a timely manner. To ensure that the relevant parties are able to interpret each audit record, a standard audit record format shall be defined and used consistently in the system. This record format shall be properly documented and made available to all relevant parties that need to handle audit records.

The SKPP also mandates that an audit record shall minimally contain the following information:

- Data and time of an event,
- Type of an event,
- Subject Identity,

- Success or failure of the event, and
- The identity of the relevant resource (where applicable)

E. SUMMARY

This chapter provided an overview of the various requirements for the LPSK audit subsystem based on the SKPP and TCX LPSK functional specification documents. A design and implementation to address the requirements is discussed in the next chapter.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. DESIGN AND IMPLEMENTATION

This chapter starts with an overview of the design of the audit subsystem components. It is followed by a discussion on the audit record format and design of the audit record generation and collection. The fourth section describes the implementation of the audit module interfaces. The last section describes the implementation of the kernel's audit buffer.

A. HIGH LEVEL DESIGN

This section presents a high level view of the LPSK audit subsystem and discusses the considerations and choices made in the design of the audit subsystem and its various components.

1. Overview

Figure 1 shows a high level overview of the interactions between the audit subsystem modules and the other system components.

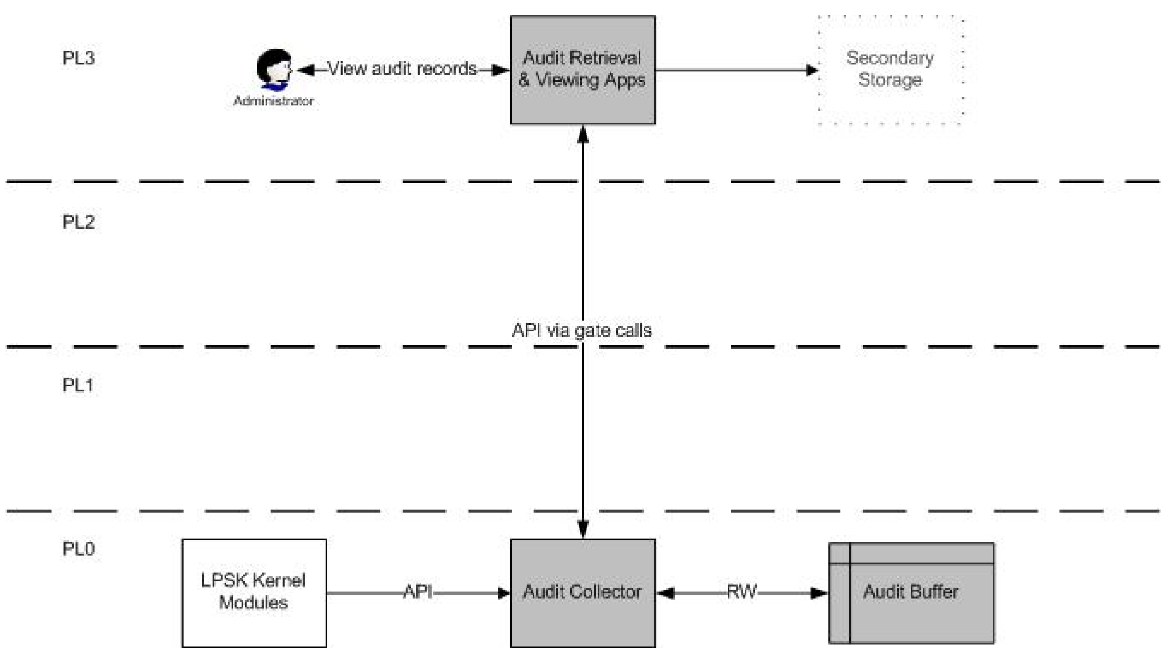

Figure 1. High Level Overview of Audit Subsystem

The LPSK platform is based on an Intel x86 processor, which consists of 4 Privilege Levels (PL), ranging from PL0, which is the most privileged, to PL3, which is the least privileged. The LPSK audit subsystem resides in PL0. The audit events are detected by other kernel modules, which then communicates relevant event information to the audit subsystem through a module interface.

The Audit Collector module receives the audit request from the kernel functions and determines whether to record the audit information based on the configured audit policy. If the event is to be audited, the Audit Collector module will format the audit record in the appropriate binary format and send it to the Audit Buffer.

The Audit Buffer provides temporary storage for the audit records before they are read by an authorized subject and transferred to a log file in a secondary storage space. Due to the limited space in the Audit Buffer, old records that have been read by an authorized subject are erased from the Audit Buffer so that memory space can be reused to store new records.

The Audit Retrieval and Viewing Application currently resides in PL3 so that it is able to make full use of the richer range of services from the underlying layers to provide a user interface to an administrator to view and manage audit records. The Retrieval and Viewing Application issues requests to the Audit Collector module which will then retrieve the oldest record from the Audit Buffer and forward it to the application for further processing.

2. Starting and Stopping the Audit Subsystem

Figure 2 shows the sequence of starting, running and stopping of the audit subsystem.

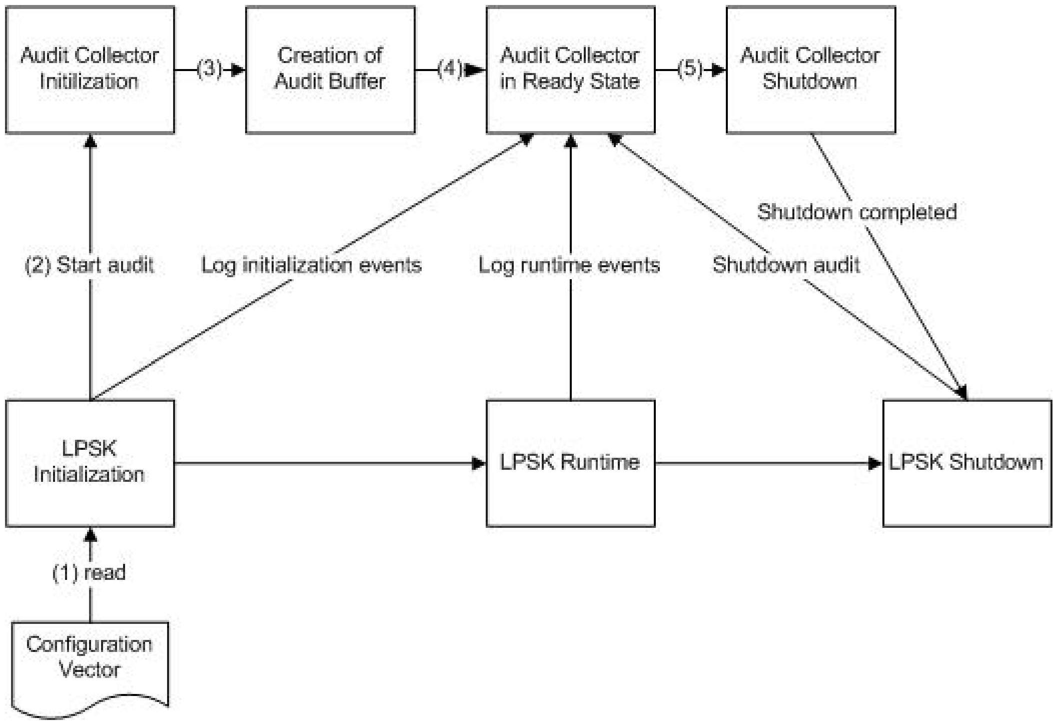

Figure 2. Audit Subsystem Life Cycle

1. The user selects a configuration vector for the LPSK. The LPSK Initializer will read the binary data of the configuration vector and configure the system state according to what is specified inside the configuration vector.

2. The LPSK Initializer will start the Audit Collector, passing configuration data to it. The configuration vector contains information such as the size of the Audit Buffer and types of events to be audited.

3. The Audit Collector initializes its internal variables and creates an audit buffer based on values specified in the configuration vector.

4. The Audit Collector will then enter into the ready state and is able to accept audit requests. Because the Audit Collector needs to start collecting records during the LPSK initialization phase, it must be started at the earliest possible stage.

5. The Audit Collector will continue to respond to audit requests throughout the entire LPSK runtime phase and during the LPSK shutdown phase. In order to

give ample time for the Audit Retrieval to retrieve all the records that are remaining in the Audit Buffer, the Auditor Collector should be among the last processes to shutdown [5]. In addition, the Audit Collector can also initiate a time delay before shutting down the audit subsystem. The value of the time delay can be configured in the configuration vector.

B. AUDIT RECORD FORMATS

A standard audit log format is important to allow the different components of a system to exchange and interpret audit records correctly. However, a survey of audit log formats used by the common operating systems shows that most of them use different proprietary formats. There is no de facto standard in the industry today. The many different types of log formats in use today include text-based, Extensible Markup Language (XML) [13], databases and binary files. Some of these formats (such as text-based and XML files) are designed to be read by humans, while others (such as databases and binary files) are not. The section below provides a discussion and comparison of the popular types of log formats. Criteria for audit format selection for the LPSK are discussed. This is followed by discussion of the LPSK audit record format.

1. Types of Audit Record Format

The various types of audit record formats are described below.

a. Syslog

Text-based formats can be in the form of a comma-separated or tab-separated text file, which can be proprietary in nature or it can follow the syslog format [14]. Syslog was initially developed as part of the Sendmail project [15] but due to its ease of use, has been widely used by many applications, especially in UNIX systems.

A syslog record consists of 3 parts. The first part contains the identity of the source and severity level. The second part contains a timestamp and the hostname while the third part contains the actual log message content. There are, however, no standards for what information is required or how the information should be formatted. It is usually just a string of text that describes what happened. While this provides the

flexibility to decide what information to include inside the content, different implementations may lead to a difficulty in interpreting the log entries. This greatly limits its potential use as a standard log format for audit log exchange among different systems.

The fact that the log content is in text format means that while they are highly readable by humans, log records cannot be easily parsed by machines. Processing and filtering of syslog records based on the attributes of the events can be challenging. A text-based format also takes up much more space compared to a binary format. This could cause significant problems for platforms with limited resources, such as handheld devices.

b. XML

XML [13] is a markup language to represent text data in a well-structured way. It is still text-based, but due to the highly structured nature of an XML document, it is meant to be both human-readable and parseable by machines. It is generally used to exchange information among different applications. The advantages of logging in XML format include ease of viewing, processing, and is well understood by many applications. There is, however, a lack of an open standard on a XML log formats. XML files also take up much more space than a binary file and require a high amount of computational resources to parse the file, making it a poor choice for platforms with limited resources.

c. Database

Logs can be stored directly into relational databases. Databases provide advanced indexing and search capabilities that text or binary log files are not able to provide. But the need for a database server means that it is highly unlikely to be used as a native log format in the kernel. It is more suitable to be used by higher level applications rather than by operating systems.

d. Binary Format

A binary format is commonly used by the major operating systems. Some of the examples are Event Log File Format [16] used by Microsoft Windows operating systems and Basic Security Module (BSM) [17] used by Solaris. The main advantage of a

binary log file is that it is small when compared to a text-based format. It is also highly flexible with regard to the type of data that it can store. Also, binary logs can be formatted in a way that allows for easy parsing by machines. The main drawback of a binary format is that it requires a log viewer application to translate the binary bits to human-readable text so that the administrator can review the logs.

2. Selection of an Audit Log Format for LPSK

In this section, the criteria for selection of the LPK audit record format are presented and audit record types are compared for suitability in the LPSK.

One of the most important factors when considering an audit log format for the LPSK is that it must be lightweight in terms of both storage and computational resources. A kernel should be small and contain only the essential services. Making the audit log small and simple also allows the deployment of the LPSK on handheld devices that have limited storage and computational resources.

Based on the survey of the different log formats, binary logs have been selected as the best choice for the LPSK because the amount of space required for collection of audit records is significantly smaller than the other log formats, and the binary format can be parsed easily. The Syslog and XML formats are attractive options for remote logging when logs need to be transferred from multiple hosts to a central log server. The binary logs collected by the LPSK can be converted to these formats at a later stage if there is such a requirement. Solaris also takes a similar approach by providing a *praudit* [17] utility to convert BSM audit records to human-readable text or XML format when required. A database can also be used to store audit records in a remote log server. Table 3 gives a comparison of the various log formats.

Table 3. Comparison of Different Audit Log Format

	Syslog	XML	Database	Binary
Human-Readable	Yes	Yes	No	No
Easily Parseable	No	Yes	Yes	Yes
Understood by multiple applications	Yes	Yes	No	No
Size	Large	Large	Depends	Small
Computational Resource Requirement	Low	High	High	Low

3. LPSK Audit Record Format

Details of the LPSK audit record format are presented in this subsection.

a. Overview

After comparing the different audit record formats, a binary based log format was chosen as the format for the LPSK. A study was conducted to determine whether any of the existing binary audit record formats can be used directly by the LPSK, but none of them is entirely suitable because most are designed to work in a specific operating system environment. For example, BSM includes classes of log records describing events related to the UNIX file system that are not relevant in the LPSK. Similarly, there are events specific to the LPSK that cannot be described by any of the existing audit record formats. Thus, there is a need to design a set of record formats specifically for the LPSK. The BSM Format was used as a reference model when

designing the audit log format for the LPSK due to the fact that it is flexible and simple to implement, and that the LPSK can benefit from its design.

b. Audit Record Structure

Each audit record represents an event that has been selected for auditing according to the configuration vector. An audit record is made up of a sequence of tokens, each describing an attribute of the event that the audit record describes. Each record begins with a header token and ends with a trailer token. There could be one or more tokens of other types in between the header and trailer tokens.

Each token starts with a one-byte token identifier, which indicates the type of attributes the token carries. There could be one or more attributes in the token depending on the token type. As a result, the length of each type of token varies. The varying token length approach of the audit record format allows a high degree of flexibility in constructing records for different types of events with different attributes. It also ensures that no space is wasted as in the case of a fixed length record that allocates the maximum amount of storage space for each record. This helps to keep the audit log small.

The following is an example of an audit record that describes the detection of a Secure Attention Key (SAK) invocation.

13 bytes	2 bytes	5 bytes
Header Token	Partition Token	Trailer Token

The header token contains the event identifier, timestamp and the length of the entire record. The event identifier indicates the type of event the record describes. The partition token contains information about the partition from which the SAK is invoked and the trailer token contains a checksum to detect accidental modifications of the audit record.

c. *Tokens*

A total of 17 tokens have been defined in order to describe the various auditable LPSK events. A header token and a trailer token will be found in all audit records. The other tokens are used when additional attributes are required in the record. Each token starts with a one-byte unique token identifier to allow the parser to know the type of data that follows. Table 4 shows the LPSK token identifiers.

Table 4. Token Identifier

Token	**Token Identifier (in hexadecimal)**
Header Token	0x00
Trailer Token	0xFF
Argument Token	0x10
Configuration Vector Token	0x11
Device Token	0x12
Dseg Token	0x13
Eventcount Token	0x14
Interrupt Token	0x15
MAC Token	0x16
Mseg Token	0x17
Partition Token	0x18
Process Token	0x19
Return Token	0x1A
Sequence Token	0x1B
Signal Token	0x1C
Subject Token	0x1D
Text Token	0x1E

The following subsections describe each token in more detail

(1) Header Token. A header token marks the beginning of each audit record. It contains the following fields in the order that they are listed.

- Header token identifier (1 byte),
- Audit record length, in bytes (2 bytes),
- Audit record structure version number (2 bytes),
- Event identifier to indicate the type of audit event
- Event modifier to provide additional information about the event (2 bytes), and
- Timestamp of the creation of the audit record (4 bytes)

1 byte	2 bytes	2 bytes	2 bytes	2 bytes	4 bytes
Token identifier	Record length	Version number	Event identifier	Event modifier	Timestamp

Figure 3. Header Token

The header token identifier is given a predefined value of 0. Audit record length stores the total number of bytes the entire record contains, including the header and the trailer tokens. As the length of each record varies, audit record length allows the parser to know how many bytes to read for the record. The version number is set aside to facilitate future modification to the record structure, so that a parser can parse newer and older record structures.

Each type of auditable event is described using a unique event identifier. The event modifier is used as an additional flag to provide more detailed descriptions of the events. For example, when recording the “successful start-up and shutdown of LPSK audit mechanism”, a modifier with a value of zero (0) indicates that it is a start-up event, while a modifier with a value of one (1) indicates that it is a shutdown event.

The LPSK uses the number of seconds since the start of the IEEE POSIX epoch [18] to keep track of date and time for its kernel. The epoch time system records date and time in terms of number of seconds elapsed since January 1 1970 00:00:00 UTC. Using 4 bytes for the timestamp means this audit record structure can keep track of time till the year 2106.

(2) Trailer Token. The trailer token marks the end of a record. It contains the following fields in the order that they are listed.

- Trailer token identifier (1 byte),
- Audit record length, in bytes (2 bytes),
- CRC32 checksum of the entire record (4 bytes)

1 byte	2 bytes	4 bytes
Token identifier	Record length	CRC32 checksum

Figure 4. Trailer Token

The record length is contained in both the header and trailer tokens. The purpose of this redundancy is to allow for the forward and backward parsing of the records. The CRC32 checksum is appended to the end of each record to support checks for accidental corruption of the record.

(3) Argument Token. The argument token contains information about argument values passed to a kernel function. It contains the following fields in the order that they are listed.

- Argument token identifier (1 byte),
- Argument identifier (1 byte),
- Argument value (4 bytes),
- Length of optional text descriptive text string (1 byte), and

- Optional text string (n bytes where $0 \le n \le 255$)

1 byte	1 byte	4 bytes	1 byte	n bytes
Token identifier	Argument identifier	Argument value	Text length	Text

Figure 5. Argument Token

A function call may contain several parameters. Thus, an audit record may also contain several argument tokens. In this case, the argument identifier indicates which parameter it corresponds to. The optional text string provides the flexibility of adding text descriptions to the arguments if needed.

(4) Configuration Vector Token. The configuration vector token contains information to identify the configuration vector used to initialize the LPSK. It contains the following fields in the order that they are listed.

- Configuration vector token identifier (1 byte),
- Text length (1 byte),
- Descriptive text (n bytes where $0 \le n \le 255$), and
- MD5 hash (16 bytes)

1 byte	1 byte	n bytes	16 bytes
Token identifier	Text length	Descriptive text	MD5 hash

Figure 6. Configuration Vector Token

The descriptive text contains the human readable description of the configuration vector. The MD5 hash value of the entire binary configuration vector provides a means to correctly identify the configuration vector that is used to initialize the LPSK.

(5) Device Token. The device token contains information about the device. It contains the following fields in the order that they are listed.

- Device token identifier (1 byte),
- Major number (1 byte),
- Minor number (1 byte),
- Type (1 byte), and
- Partition ID (1 byte)

1 byte	1 byte	1 byte	1 byte	1 byte
Token identifier	Major number	Minor number	Type	Partition ID

Figure 7. Device Token

The major number indicates the device category. The minor number refers to the specific instantiation of a device. The type attributes can be either CONTROL or DATA depending on how the device was accessed. The partition ID is the home partition of the device.

(6) Dseg Token. The dseg token contains information about the data segments. It contains the following fields in the order that they are listed.

- Dseg token identifier (1 byte),
- Privilege level assigned to dseg (1 byte),
- Partition ID (1 byte),
- Path length (1 byte), and
- Path (n bytes)

1 byte	1 byte	1 byte	1 byte	n bytes
Token identifier	Privilege level	Partition ID	Path length	Path

Figure 8. Dseg Token

A *dseg* is a data segment in a process' address space that is initialized from a secondary storage segment. The maximum number of dsegs defined in the LPSK specification is 64. Each dseg is identified by a unique identifier. The privilege level is the Intel PL to which the dseg was allocated during initialization. The partition ID is the home partition of dseg. The path attribute contains the path to the secondary storage segment, while the path length indicates the length of the path.

(7) Eventcount Token. The evencount token contains information about the eventcount. It contains the following fields in the order that they are listed.

- Eventcount token identifier (1 byte),
- Eventcount ID, and
- Eventcount value

1 byte	1 byte	4 bytes
Token identifier	Eventcount ID	Eventcount value

Figure 9. Eventcount Token

The eventcount is used for inter-process synchronization. A maximum of 64 platform-wide eventcounts is possible, and each has a unique eventcount ID and stores a 32-bit number. The eventcount value is the value of the eventcount at the time of the audited event.

(8) Interrupt Token. The interrupt token contains information about the interrupts. The Intel x86 architecture provides a total of 256 interrupts, where

each is identified by a unique interrupt number. The token contains the following fields in the order that they are listed.

- Interrupt token identifier (1 byte), and
- Interrupt number (1 byte)

1 byte	1 byte
Token identifier	Interrupt number

Figure 10. Interrupt Token

(9) MAC Token. The MAC Token contains information about the Media Access Control (MAC) address. It contains the following fields in the order that they are listed.

- MAC token identifier (1 byte), and
- MAC address (6 bytes)

1 byte	6 bytes
Token identifier	MAC address

Figure 11. MAC Token

(10) Mseg Token. The Mseg token contains information about the memory segment. It contains the following fields in the order that they are listed.

- Mseg token identifier (1 byte),
- Mseg identifier (1byte),
- Size of the mseg (4 bytes),
- Privilege level assigned to mseg (1 byte), and
- Partition ID (1 byte)

1 byte	1 byte	4 bytes	1 byte	1 byte
Token identifier	Mseg identifier	Size	Privilege level	Partition ID

Figure 12. Mseg Token

An *mseg* is an Intel x86 data segment that is created in a process' address space. The maximum number of msegs defined in the LPSK specification is 32. Each mseg is identified by a unique identifier. The size is specified during initialization. The privilege level is the Intel PL to which the mseg was allocated during initialization. The partition ID refers to the home partition that the mseg belongs to.

(11) Partition Token. A partition token contains information to identify a partition. It contains the following fields in the order that they are listed.

- Partition token identifier (1 byte), and
- Partition ID (1 byte)

1 byte	1 byte
Token identifier	Partition ID

Figure 13. Partition Token

The maximum number of partitions defined in the LPSK specification is 256. Each partition is identified by a unique identifier called the partition ID.

(12) Process Token. A process token contains information to identify a process. It contains the following fields in the order that they are listed.

- Process token identifier (1 byte),
- Partition ID (1 byte), and
- Process Identifier (1 byte)

1 byte	1 byte	4 bytes
Token identifier	Partition ID	Process identifier

Figure 14. Process Token

(13) Return Token. The return token contains the return status of a kernel function call. It contains the following fields in the order that they are listed.

- Return token identifier (1 byte), and
- Return value (4 bytes)

1 byte	4 bytes
Token identifier	Return value

Figure 15. Return Token

(14) Sequencer Token. The sequencer token contains information to identify a sequencer. It contains the following fields in the order that they are listed.

- Sequencer token identifier (1 byte),
- Sequencer identifier (1 byte), and
- Sequencer value (4 bytes)

1 byte	1 byte	4 bytes
Token identifier	Sequencer identifier	Sequencer value

Figure 16. Sequence Token

A sequencer is used for inter-process synchronization. A maximum of 64 platform-wide sequencers is possible in the LPSK. Each is identified by a unique

identifier. The sequencer value is the value of the sequencer at the time of the audited event.

(15) Signal Token. The signal token contains information to identify a signal channel. It contains the following fields in the order that they are listed.

- Signal token identifier (1 byte), and
- Signal channel identifier (1 byte)

1 byte	1 byte
Token identifier	Signal channel identifier

Figure 17. Signal Token

A signal is an abstract communication mechanism implemented by the LPSK to allow a subject to communicate with another subject via the recipient's signal channel. Each subject can have a maximum of 32 signal channels. A signal token is usually used together with subject tokens to provide information about the sender and receiver of a signal.

(16) Subject Token. The subject token contains information to identify a subject. It contains the following fields in the order that they are listed.

- Subject token identifier (1 byte),
- Partition ID (1 byte),
- Process identifier (4 bytes), and
- Hardware Privilege level (1 byte)

1 byte	1 byte	4 bytes	1 byte
Token identifier	Partition ID	Process Identifier	Hardware Privilege level

Figure 18. Subject Token

The hardware privilege level is the Intel PL to which the subject was allocated during initialization.

(17) Text Token. The text token describes a text string. It contains the following fields in the order that they are listed.

- Text token identifier (1 byte),
- Length of the text string (1 byte), and
- Text string (n bytes where $0 \leq n \leq 255$)

1 byte	1 byte	n bytes
Token identifier	Length of text	Text

Figure 19. Text Token

4. Event Classes and Identifier

There are altogether 45 auditable events, and each of them has a unique two-byte event identifier. They are categorized into a number of different classes for ease of management. The first byte of the event identifier indicates the class and the second byte indicates the event number within the class. For example, the event "successful completion of LPSK initialization" belongs to the *Initilization* class and has a class identifier of 1. As it is the 5th event within the class, it is being assigned an event identifier of 0x0105 (in hexadecimal). Table 5 provides information about the event classes.

Table 5. Event Classes

Class Name	**Description**	**Class Identifier (in hexadecimal)**
Initialization	Events that occur during the initialization phase of the LPSK	0x01
System	System-wide events that occur during the runtime phase of the LPSK	0x02
Device	Events related to devices	0x03
Process	Events related to process management	0x04
Memory	Events related to memory management	0x05
Synchronization	Events related to resources used for synchronization such as eventcount and sequencer.	0x06

Table 6 shows the list of events in the *Initialization* class, their corresponding event identifiers and sequence of tokens used to construct them.

Table 6. Events in Initialization Class

Events in Initialization Class	**Sequence of Tokens**	**Event ID (in hex)**
Identifying information about configuration vector	header conf. vector trailer	0x0101
Unsuccessful binding of security attributes to individual partitions, subjects, and non-subject exported resources.	Headertext trailer	0x0102
The assignment of a default value to the configuration data during LPSK initialization.	header text trailer	0x0103
The detection during LPSK initialization of an invalid value or set of values in a binary configuration vector.	headertext trailer	0x0104
The successful completion of LPSK initialization	header trailer	0x0105

Table 7 shows the list of events in the *System* class, their corresponding event identifiers and the sequence of tokens used to construct them.

Table 7. Events in System Class

Events in System Class	Sequence of Tokens	Event ID (in hex)
Successful start-up and shutdown of the LPSK audit mechanism by the LPSK Initializer. (Event modifier: 0 indicates a start-up, and 1 indicates a shutdown)	header trailer	0x0201
Actions taken because of a failure of an LPSK self-test. This will result in a halt of the platform.	header text trailer	0x0202
All requests for a configuration change.	header subject argument return trailer	0x0203
The success of each startup of the LPSK.	header trailer	0x0204
A failure of an LPSK self test.	header text trailer	0x0205
Any detected loss of secure state.	header text trailer	0x0206

Action taken to attempt to recover the LPSK to a secure state.	header text trailer	0x0207
The inability of the LPSK to return to a secure state after failure of a security function.	header text trailer	0x0208
Changes to the LPSK time source.	header argument subject return trailer	0x0209
Detection of a SAK invocation.	header partition trailer	0x020A
The shutdown, powerdown or halt of a platform. (Event modifier: 0 indicates a shutdown, 1 indicates a powerdown and 2 indicates a halt)	header subject trailer	0x020B

Table 8 shows the list of events in the *Device* class, their corresponding event identifiers and sequence of tokens used to construct them.

Table 8. Events in Device Class

Events in Device Class	**Sequence of Tokens**	**Event ID (in hex)**
When a read operation of a particular device is requested (success, failure, or both)	header subject device return trailer	0x0301
When a write operation of a particular device is requested (success, failure, or both)	header subject device return trailer	0x0302
When a read meta-data operation for a particular device is requested (success, failure, or both)	header subject device argument return trailer	0x0303
When a write meta-data operation for a particular device is requested (success, failure, or both)	header subject	0x0304

	device argument return trailer	
Duplicated MAC address	header partition mac trailer	0x0305
When a device read is requested by a particular subject. (success, failure, or both)	header subject device return trailer	0x0306
When a device write is requested by a particular subject. (success, failure, or both)	header subject device return trailer	0x0307
When a device configuration is requested by a particular subject. (success, failure, or both)	header subject device argument return	0x0308

	trailer	

Table 9 shows the list of events in the *Process* class, their corresponding event identifiers and sequence of tokens used to construct them.

Table 9. Events in the Process Class

Events in Process Class	**Sequence of Tokens**	**Event ID (in hex)**
The success or failure of starting a process.	header process trailer	0x0401
The termination of a process.	header process trailer	0x0402
When a signal is sent by a particular subject (success, failure, or both)	header subject (sender) signal subject (recipient) return trailer	0x0403
When a signal is received by a particular subject	header subject (recipient) signal subject (sender)	0x0404

	return trailer	
When a software interrupt is invoked by a particular subject	header subject interrupt return trailer	0x0405

Table 10 shows the list of events in the *Memory* class, their corresponding event identifiers and the sequence of tokens used to construct them.

Table 10. Events in the Memory Class

Events in Memory Class	**Sequence of Tokens**	**Event ID (in hex)**
Attempt to swapin a dseg exceeded memory quota.	header subject dseg trailer	0x0501
Attempt to create an mseg exceeded memory quota.	header subject mseg trailer	0x0502
When a particular segment is swapped in (success, failure, or both)	header subject dseg	0x0503

	return trailer	
When a particular segment is flushed (success, failure, or both)	header subject dseg return trailer	0x0504
When a particular segment is swapped out (success, failure, or both)	header subject dseg return trailer	0x0505
When a mseg is created	header subject mseg return trailer	0x0506

Table 11 shows the list of events in the *Synchronization* class, their corresponding event identifiers and the sequence of tokens used to construct them.

Table 11. Events in the Synchronization Class

Events in Synchronization Class	**Sequence of Tokens**	**Event ID (in hex)**

When a process awakes from an await on an eventcount (success, failure, or both)	header subject eventcount return trailer	0x0601
When an advance of an eventcount is requested (success, failure, or both)	header subject eventcount return trailer	0x0602
When a read of an eventcount is requested (success, failure, or both)	header subject eventcount return trailer	0x0603
When an awake of an eventcount is requested (success, failure, or both)	header subject eventcount return trailer	0x0604
When a wakeup on an eventcount is requested (success, failure, or both)	Header subject eventcount	0x0605

	return trailer	
When a ticket of a sequencer is requested (success, failure, or both)	header subject sequencer return trailer	0x0606
When the read of an eventcount is requested by a particular subject (success, failure, or both)	header subject eventcount return trailer	0x0607
When an advance of an eventcount is requested by a particular subject (success, failure, or both)	header subject eventcount return trailer	0x0608
When an await on an eventcount is requested by a particular subject (success, failure, or both)	header subject eventcount return trailer	0x0609
When the ticket of a sequencer is requested by a	header	0x060A

particular subject (success, failure, or both)	subject sequencer return trailer	

C. DESIGN OF AUDIT GENERATION AND COLLECTION

An overview of the processes involved in the generation and collection of audit records is presented. This is followed by a discussion of the design considerations in the proposed workflow.

1. Overview

The following is the flow of events during the audit generation and collection phase:

- A kernel module reaches a potential audit event generation point, usually just before the *return* statement inside the relevant kernel function. It invokes a function call to the audit subsystem to check whether the audit is enabled. The audit subsystem replies with a predetermined return value.
- If audit is disabled, the kernel module will not perform any audit generation operations. If audit is enabled, the kernel module will gather the necessary event information and send it to the audit subsystem through a function call.
- The audit subsystem will format the event information into the correct token format, append header and trailer tokens, and write the record into the Audit Buffer.

2. Determination of Auditable Events

The configuration vector provides the flexibility for an administrator to customize the audit policy according to operational needs. The administrator can decide which audit

events are to be generated and written to the audit logs. Thus, at the audit generation and collection phase, there must also be a mechanism for the system to perform a check of the audit policy to decide whether the event should be audited.

A few approaches have been considered:

1. The kernel modules are individually responsible for storing audit policy related to their functionality, which allows the individual modules to determine whether a potential event is auditable before invoking any function calls to the audit subsystem. However, this means that potential audit event decision points are spread throughout the kernel code. The lack of a common module to perform the checking spreads the audit policy across many modules and may result in difficulty in management of the audit policy code.

2. Kernel modules will always send all audit event information to the audit subsystem. The audit subsystem is responsible for determining whether the event is auditable. The advantage of this approach is the ease of management of code as all audit configurations are maintained within the audit module. However, there could be a potential performance issue as all event information will be sent to the audit subsystem regardless of whether the events should be audited or not, or whether auditing has been disabled completely per the configuration vector.

3. The audit subsystem provides an interface to other kernel modules to allow them to check whether a particular event should be audited before sending all the event information. However, in this case, the audit subsystem still needs most of the event information to decide whether the event is auditable. Ultimately, there might not be any significant improvement over Option 2.

4. Audit subsystem provides an interface to other kernel modules to allow them to check whether audit is enabled. This helps to reduce the amount of unnecessary event data transferred in the event that audit is disabled.

Option 4 combined with Option 2 were selected and implemented in this research because it provides a balanced approach by not requiring kernel modules to perform their

own audit policy checking while still eliminating unnecessary function calls when audit is disabled. Future work is needed to assess the performance impact of Option 2 before deciding the best way to perform the audit policy checking.

3. Audit Module Interfaces

In order for the LPSK modules to communicate with the audit subsystems, a set of application programming interfaces (API) must be provided by the audit subsystem. Ideally, one common interface can be used by all LPSK modules to send event information to the audit subsystem. However, that is impractical because the relevant information differs from event to event. Such a one-size-fits-all interface would require the interface to support all possible parameters, even though only a few would be used for most events. A more practical approach is to provide a separate interface for each type of event. Detailed discussions of the implementation of the interfaces are presented in Section 4.

Even with a separate interface for each type of event, the parameter list can still be very long for some events. For easy management, a structure type is defined for each type of token and the event information is encapsulated inside the token structure type. Pointers to the structure types are passed as parameters to the audit subsystem. This helps to keep the list of parameters small. The use of typed structures enhances understandability and makes it easier to make amendments in the future.

D. AUDIT MODULE INTERFACES IMPLEMENTATION

The LPSK audit module prototype was developed using the C programming language. The implementation of the interfaces is discussed next.

1. Interfaces to Kernel Modules

Table 12 shows the structure types defined for the different types of tokens. Event information is stored in these structure types and pointers to them are passed as parameters to the audit subsystem interfaces.

Table 12. Structure Types for Tokens

Structure Type	Descriptions
typedef struct { unsigned char arg_id; unsigned int arg_val; unsigned char txt_len; unsigned char txt[256]; } audit_token_argument_t;	Structure type for argument token • *arg_id* refers to the argument identifier • *arg_val* refers to the argument value • *txt_len* refers to the length of optional descriptive text string • *txt* refers to the optional descriptive text
typedef struct { unsigned char txt_len; unsigned char txt[256]; unsigned int size; unsigned char md5hash[16]; } audit_token_cv_t;	Structure type for configuration vector token • *txt_len* refers to the length of the configuration vector descriptive text • *txt* refers to the configuration vector descriptive text • *size* refers to the size of the configuration vector • *md5hash* refers to the MD5 hash value of the binary configuration vector
typedef struct { unsigned char major; unsigned char minor; unsigned char type; unsigned char part_id; } audit_token_device_t;	Structure type for device token • *major_num* refers to the major number of the device • *minor_num* refers to the minor number of the device • *type* refers to the type of device • *part-id* refers to the partition ID of the device
typedef struct {	Structure type for dseg token

unsigned char pl; unsigned char part_id; unsigned char path_len; unsigned char path[256]; } audit_token_dseg_t;	• *pl* refers to the privilege level of dseg • *part_id* refers to the home partition ID of dseg • *path_len* refers to the length of the dseg path • *path* refers to the dseg path
typedef struct { unsigned char ec_id; unsigned int ec_value; } audit_token_eventcount_t;	Structure type for eventcount token • *ec_id* refers to the eventcount ID • *ec_value* refers to the value of the eventcount
typedef struct { unsigned char int_num; } audit_token_interrupt_t;	Structure type for interrupt token • *int_num* refers to the interrupt number
typedef struct { unsigned char mac_addr[6]; } audit_token_mac_t;	Structure type for MAC token • *mac_addr* refers to the MAC address
typedef struct { unsigned char mseg_id; unsigned int size; unsigned char pl; unsigned char part_id; } audit_token_mseg_t;	Structure type for mseg token • *mseg_id* refers to the mseg identifier • *size* refers to the size of the mseg • *pl* refers to the privilege level of mseg • *part_id* refers to the home partition ID of mseg
typedef struct { unsigned char part_id; } audit_token_partition_t;	Structure type for partition token • *part_id* refers to the partition ID

typedef struct { unsigned char part_id; unsigned char proc_id; } audit_token_process_t;	Structure type for process token • *part_id* refers to partition ID of the process • *proc_id* refers to the process ID of the process
typedef struct { unsigned int ret_val; } audit_token_return_t;	Structure type for return token • *ret_val* refers to the return value of a function call
typedef struct { unsigned char seq_id; unsigned int seq_value; } audit_token_sequencer_t;	Structure type for sequencer token • *seq_id* refers to the sequencer ID • *seq_value* refers to the value of the sequencer
typedef struct { unsigned char sig_channel; } audit_token_signal_t;	Structure type for signal token • *sig_channel* refers to the signal channel
typedef struct { unsigned char part_id; unsigned int proc_id; unsigned char pl; } audit_token_subject_t;	Structure type for subject token • *part_id* refers to the partition ID of the subject • *proc_id* refers to the process ID of the subject • *pl* refers to the privilege level of the subject
typedef struct { unsigned char txt_len; unsigned char txt[256]; } audit_token_text_t;	Structure type for subject token • *txt_len* refers to the length of the text string • *txt* refers to the text string

Table 13 describes a list of interfaces that are provided to allow LPSK modules to communicate with the audit subsystem.

Table 13. Interfaces Provided to LPSK Modules

Interfaces	Description
unsigned int audit_enabled(void)	Returns TRUE if audit is enabled and returns FALSE if not.
unsigned int audit_write_INI_cv(audit_token_cv_t *cv_tokptr);	Generate an audit record with identifying information about the configuration vector. The input parameter is a pointer to a configuration vector token type that contains information identifying the configuration vector.
unsigned int audit_write_INI_bind(audit_token_text_t *text_tokptr);	Generate an audit record for the unsuccessful binding of security attributes to individual partitions, subjects, and non-subject exported resources. The input parameter is a pointer to a text token type that describes the event.
unsigned int audit_write_INI_assign(audit_token_text_t *text_tokptr);	For some fields in the configuration vector, a declared value is optional. When an optional value is not given, the LPSK platform is required to use a default value for the duration of an operational mode. Generate an audit

	record when such assignments of default values during initialization occurred. The input parameter is a pointer to a text token type that describes the event.
unsigned int audit_write_INI_invalid(audit_token_text_t *text_tokptr);	Generate an audit record for the detection during LPSK initialization of an invalid value or set of values in a binary configuration vector. The input parameter is a pointer to a text token type that describes the event.
unsigned int audit_write_INI_complete(void);	Generate an audit record for the successful completion of LPSK initialization
unsigned int audit_write_SYS_auditstart(unsigned short evt_mod);	Generate an audit record for the successful start-up and shutdown of the LPSK audit mechanism by the LPSK Initializer. An input argument of 0 indicates a start-up, and 1 indicates a shutdown
unsigned int audit_write_SYS_actiontest(audit_token_text_t *text_tokptr);	Generate an audit record for the actions taken because of a failure of an LPSK self-test. The input parameter is a pointer to the text token type that describes the actions taken.
unsigned int audit_write_SYS_configchange(	Generate an audit record for all requests

audit_token_subject_t *sub_tokptr audit_token_argument_t *arg_tok, audit_token_return_t *return_tok);	for a configuration change. The input parameters are pointers to a subject token type and an argument token type that describes the argument provided for the configuration change and a return token type that indicates the return value.
unsigned int audit_write_SYS_lpskstart(void);	Generate an audit record for the success of each startup of the LPSK.
unsigned int audit_write_SYS_failtest(audit_token_text_t *text_tokptr);	Generate an audit record for the failure of an LPSK self test. The input parameter is a pointer to the text token type that describes the event.
unsigned int audit_write_SYS_loss(audit_token_text_t *text_tokptr);	Generate an audit record for any detected loss of secure state. The input parameter is a pointer to the text token type that describes the event.
unsigned int audit_write_SYS_recover(audit_token_text_t *text_tokptr);	Generate an audit record for an action taken to attempt to recover the LPSK to a secure state. The input parameter is a pointer to the text token type that describes the action taken.
unsigned int audit_write_SYS_failsecure(audit_token_text_t *text_tokptr);	Generate an audit record for the inability of the LPSK to return to a secure state

	after failure of a security function. The input parameter is a pointer to the text token type that describes the event.
unsigned int audit_write_SYS_time(audit_token_subject_t *sub_tokptr, audit_token_argument_t *arg_tokptr, audit_token_return_t *return_tokptr);	Generate an audit record for changes to the LPSK time source. The input parameters are pointers to a subject token type, an argument token type that describes the argument provided for the change to the time source, and a return token type that indicates the return value.
unsigned int audit_write_SYS_sak(audit_token_partition_t *part_tokptr);	Generate an audit record for the detection of a SAK invocation. The input parameter is a pointer to the partition token type that describes the partition.
unsigned int audit_write_SYS_shut(unsigned short evt_mod);	Generate an audit record for the shutdown, powerdown or halt of a platform. An input argument of 0 indicates a shutdown, 1 indicates a powerdown and 2 indicates a halt.
unsigned int audit_write_DEV_read(audit_token_subject_t *sub_tokptr, audit_token_device_t *device_tok, audit_token_return_t *return_tok);	Generate an audit record when a read operation of a particular device is requested. The input parameters are pointers to a subject token type, a device token type and a return token type.

unsigned int audit_write_DEV_write(audit_token_subject_t *sub_tokptr, audit_token_device_t *device_tok, audit_token_return_t *return_tok);	Generate an audit record when a write operation of a particular device is requested. The input parameters are pointers to a subject token type, a device token type and a return token type.
unsigned int audit_writeDEV_metaread(audit_token_subject_t *sub_tokptr, audit_token_device_t *device_tok, audit_token_arg_t *arg_tok, audit_token_return_t *return_tok);	Generate an audit record when a read meta-data operation for a particular device is requested. The input parameters are pointers to a subject token type, a device token type, an argument token type and a return token type.
unsigned int audit_write_DEV_metawrite(audit_token_subject_t *sub_tokptr, audit_token_device_t *device_tok, audit_token_arg_t *arg_tok, audit_token_return_t *return_tok);	Generate an audit record when a write meta-data operation for a particular device is requested. The input parameters are pointers to a subject token type, a device token type, an argument token type and a return token type.
unsigned int audit_write_DEV_mac(audit_token_partition_t *part_tokptr, audit_token_mac_t *mac_tok);	Generate an audit record when duplicate MAC addresses are detected. The input parameters are pointers to a partition token type and a mac token type that contains the MAC address.

unsigned int audit_write_DEV_subread(audit_token_subject_t *sub_tokptr, audit_token_device_t *device_tok, audit_token_return_t *return_tok);	Generate an audit record when a device read is requested by a particular subject. The input parameters are pointers to a subject token type, a device token type and a return token type.
unsigned int audit_write_DEV_subwrite(audit_token_subject_t *sub_tokptr, audit_token_device_t device_tok, audit_token_return_t return_tok);	Generate an audit record when a device write is requested by a particular subject. The input parameters are pointers to a subject token type, a device token type and a return token type.
unsigned int audit_writeDEV_subconf(audit_token_subject_t *sub_tokptr, audit_token_device_t device_tok, audit_token_arg_t arg_tok, audit_token_return_t return_tok);	Generate an audit record when a device configuration is requested by a particular subject. The input parameters are pointers to a subject token type, a device token type, an argument token type and a return token type.
unsigned int audit_write_PRO_start(audit_token_process_t *proc_tokptr);	Generate an audit record for the success or failure of starting a process. The input parameter is a pointer to a process token type.
unsigned int audit_write_PRO_terminate(audit_token_process_t *proc_tokptr);	Generate an audit record for the termination of a process. The input parameter is a pointer to a process token type.
unsigned int audit_write_PRO_sigsent(	Generate an audit record when a signal

audit_token_subject_t *subject_tok, audit_token_signal_t *signal_tok, audit_token_subject_t *subject_tok, audit_token_return_t *return_tok);	is sent by a particular subject. The input parameters are pointers to a sender subject token type, a signal token type, a recipient subject token type and a return token type.
unsigned int audit_write_PRO_sigrecv(audit_token_subject_t *subject_tok, audit_token_signal_t *signal_tok, audit_token_subject_t *subject_tok, audit_token_return_t *return_tok);	Generate an audit record when a signal is received by a particular subject. The input parameters are pointers to a recipient subject token type, a signal token type, a sender subject token type and a return token type.
unsigned int audit_write_PRO_interrupt(audit_token_subject_t *subject_tok, audit_token_interrupt_t *interrupt_tok, audit_token_return_t *return_tok);	Generate an audit record when a software interrupt is invoked by a particular subject. The input parameters are pointers to a subject token type, an interrupt token type, and a return token type.
unsigned int audit_write_MEM_dsegexceed(audit_token_subject_t *subject_tok, audit_token_dseg_t *dseg_tok);	Generate an audit record when an attempt to swapin a dseg exceeded memory quota. The input parameters are pointers to a subject token type, and a dseg token type.
unsigned int audit_write_MEM_msegexceed(audit_token_subject_t *subject_tok, audit_token_mseg_t *mseg_tok);	Generate an audit record when an attempt to create a mseg exceeds the memory quota. The input parameters are

	pointers to a subject token type, and a mseg token type.
unsigned int audit_write_MEM_swapin(audit_token_subject_t *subject_tok, audit_token_dseg_t *dseg_tok, audit_token_return_t *return_tok);	Generate an audit record when a particular segment is swapped in. The input parameters are pointers to a subject token type, a dseg token type and a return token type.
unsigned int audit_write_MEM_flush(audit_token_subject_t *subject_tok, audit_token_dseg_t *dseg_tok, audit_token_return_t *return_tok);	Generate an audit record when a particular segment is flushed. The input parameters are pointers to a subject token type, a dseg token type and a return token type.
unsigned int audit_write_MEM_swapout(audit_token_subject_t *subject_tok, audit_token_dseg_t *dseg_tok, audit_token_return_t *return_tok);	Generate an audit record when a particular segment is swapped out. The input parameters are pointers to a subject token type, a dseg token type and a return token type.
unsigned int audit_write_MEM_msegcreate(audit_token_subject_t *subject_tok, audit_token_mseg_t *mseg_tok, audit_token_return_t *return_tok);	Generate an audit record when a mseg is created. The input parameters are pointers to a subject token type, a mseg token type and a return token type.
unsigned int audit_write_SYN_procawait(audit_token_subject_t *subject_tok, audit_token_eventcount_t *evtcnt_tok, audit_token_return_t *return_tok);	Generate an audit record when a process awakes from an await on an eventcount. The input parameters are pointers to a subject token type, an eventcount token

	type and a return token type.
unsigned int audit_write_SYN_ecadvance(audit_token_subject_t *subject_tok, audit_token_eventcount_t *evtcnt_tok, audit_token_return_t *return_tok);	Generate an audit record when an advance of an eventcount is requested. The input parameters are pointers to a subject token type, an eventcount token type and a return token type.
unsigned int audit_write_SYN_ecread(audit_token_subject_t *subject_tok, audit_token_eventcount_t *evtcnt_tok, audit_token_return_t *return_tok);	Generate an audit record when a read of an eventcount is requested. The input parameters are pointers to a subject token type, an eventcount token type and a return token type.
unsigned int audit_write_SYN_ecawake(audit_token_subject_t *subject_tok, audit_token_eventcount_t *evtcnt_tok, audit_token_return_t *return_tok);	Generate an audit record when an awake of an eventcount is requested. The input parameters are pointers to a subject token type, an eventcount token type and a return token type.
unsigned int audit_write_SYN_ecwakeup(audit_token_subject_t *subject_tok, audit_token_eventcount_t *evtcnt_tok, audit_token_return_t *return_tok);	Generate an audit record when a wakeup on an eventcount is requested. The input parameters are pointers to a subject token type, an eventcount token type and a return token type.
unsigned int audit_write_SYN_seqticket(audit_token_subject_t *subject_tok, audit_token_sequencer_t *seq_tok, audit_token_return_t *return_tok);	Generate an audit record when the ticket of a sequencer is requested by a particular subject. The input parameters are pointers to a subject token type, a

	sequencer token type and a return token type.
unsigned int audit_write_SYN_subecread(audit_token_subject_t *subject_tok, audit_token_eventcount_t *evtcnt_tok, audit_token_return_t *return_tok);	Generate an audit record when a read of an eventcount is requested by a particular subject. The input parameters are pointers to a subject token type, an eventcount token type and a return token type.
unsigned int audit_write_SYN_subecadvance(audit_token_subject_t *subject_tok, audit_token_eventcount_t *evtcnt_tok, audit_token_return_t *return_tok);	Generate an audit record when an advance of an eventcount is requested by a particular subject. The input parameters are pointers to a subject token type, an eventcount token type and a return token type.
unsigned int audit_write_SYN_subecawait(audit_token_subject_t *subject_tok, audit_token_eventcount_t *evtcnt_tok, audit_token_return_t *return_tok);	Generate an audit record when an await on an eventcount is requested by a particular subject. The input parameters are pointers to a subject token type, an eventcount token type and a return token type.
unsigned int audit_write_SYN_subseqticket(audit_token_subject_t *subject_tok, audit_token_sequencer_t *seq_tok, audit_token_return_t *return_tok);	Generate an audit record when the ticket of a sequencer is requested by a particular subject. The input parameters are pointers to a subject token type, a sequencer token type and a return token type.

A selection process was performed on the 46 auditable events to identify those that are potentially implementable in the current LPSK prototype and those events that are likely to occur during the execution of each process. Ten high priority interfaces were selected and fully implemented in the LPSK audit subsystem prototype. They include the following:

- audit_write_INI_cv
- audit_write_INI_complete
- audit_write_SYS_auditstart
- audit_write_SYS_lpskstart
- audit_write_SYS_sak
- audit_write_MEM_swapin
- audit_write_MEM_msegcreate
- audit_write_SYN_ecawait
- audit_write_SYN_procawake
- audit_write_SYM_seqticket

2. Exported LPSK Audit Interfaces

Table 14 describes the interfaces that are provided to allow a non-kernel audit retrieval application to communicate with the audit subsystem.

Table 14. Exported LPSK Audit Interfaces

Interfaces	**Description**
unsigned int audit_read_next(unsigned short max_len, unsigned char *buffer, unsigned short *num_requested, unsigned short *num_read);	Reads the oldest records from the audit buffer and places them into the output parameter *buffer* (The number of records requested is indicated via *num_requested,* and the size of the *buffer* is indicated in

	max_len). Returns the number of records placed inside *buffer* through *num_read*. If there are fewer than *num_requested* records in the Audit Buffer, then the available records are put into the *buffer*, and no error is returned. If there are *num_requested* records, but they would not all fit into *buffer*, then those that will fit will be put into *buffer*, and no error is returned. If there are no audit records to be obtained, then num_read is set to to indicate that the buffer is empty.
unsigned int audit_read_buffer_size(unsigned int *buffer_size);	Reads the audit buffer size and places it into the output parameter *buffer_size*.
unsigned int audit_read_num_rec(unsigned int *num_rec);	Returns the number of records in the audit buffer and places it into the output parameter *num_rec*.
Unsigned int audit_read_num_generated(unsigned int *num_generated)	Returns the total number of records generated during the current operational mode in the output parameter *num_generated*. This number will wrap-around to zero if more than 2^{32} audit records are generated in an operational mode.
unsigned int audit_read_num_overwritten(unsigned int *num_overwritten);	Reads the number of records overwritten and places it into the output parameter

	num_overwritten. This represents the number of audit records that have been overwritten during the current operational mode because the buffer was full when a new record was generated. This number will wrap-around to zero if more than 2^{32} audit records are overwritten in an operational mode.

E. AUDIT BUFFER IMPLEMENTATION

Audit records collected by the Audit Collector are first stored in an Audit Buffer, then they may be read by an authorized Audit Retrieval subject, which may save them on a secondary storage device. Because the size of the Audit Buffer is limited, records that have been read by the Audit Retrieval subject need to be deleted from the Audit Buffer to free up space for new records.

Because records will be deleted as they are read, and new records will be added as space allows, it was determined that the Audit Buffer should be implemented using the abstract data type of a circular buffer. A circular buffer is a First-In-First-Out (FIFO) queue which means that the oldest record will be read first. The size of the Audit Buffer is specified in the configuration vector and a memory segment of that size is allocated to the Audit Buffer during the audit subsystem initialization phase.

The circular buffer is implemented as an array of bytes inside the memory segment. Two indices are used to keep track of where the oldest record starts, and where the newest record ends: *first* and *last*. These two variables mark the start and end of the queue respectively. A new audit record is added to the end of the queue, at the location referenced by the *last* index, which is then incremented to point to the end of the new record. When the *last* index reaches the end of the allocated memory segment, it will

“wrap around” and move to the beginning of the memory segment. An audit record is read from the start of the queue marked by the *first* index, which is then modified to point to the next record in the queue.

When the audit retrieval application attempts to read an empty buffer, i.e. when *first* and *last* indices point to the same location, the audit subsystem will return zero bytes of data read to the calling application to indicate that audit buffer is empty. When the LPSK modules attempt to write to a full buffer, i.e. when incrementing *last* index will cause it to point to the same location as *first* index, the audit subsystem will either overwrite the oldest record, halt or shutdown. The behavior can be configured in the configuration vector.

When the audit subsystem overwrites an audit record, it will first increment the *first* index by the amount equal to the length of the oldest record and thus makes it point to the second oldest record. In this way, it effectively discards the oldest record and allows a new record to overwrite the space used by the oldest record. It is possible that the newest record is larger than the oldest record, which may cause more than one record to be overwritten.

Two more variables, *record_num* and *overwritten_num*, are used to keep track of the number of records in the buffer and the number of records that have been overwritten respectively. They are updated whenever records are read, written or overwritten. The audit retrieval application can query the audit subsystem for the values of these variables through kernel APIs. Each variable is stored using a 32-bit unsigned integer, which will roll over to zero when the maximum value is reached. It is the responsibility of the audit retrieval application to take the necessary actions to deal with rollover.

V. TESTING

This chapter consists of two parts: The first part describes the developmental testing of the individual audit subsystem interfaces. The second part describes the acceptance testing of the entire audit subsystem to meet the requirements stipulated in Chapter III.

A. DEVELOPMENTAL TESTING

The purpose of the developmental testing is to ensure that each interface of the LPSK audit subsystem behaves in the way intended by design.

1. Testing of Interfaces to Kernel Modules

Tables 15 through 25 show the test suite for the APIs provided by the audit subsystem to other LPSK modules to allow them to write audit records to the Audit Buffer. Eleven of the internal APIs are implemented for this research. The functions were tested independently of each other after the initialization of the audit subsystem. Test code was inserted into the LPSK modules to invoke the functions using different input arguments and sometimes under different conditions. Debugging messages were generated to provide a mean of verifying the outcome.

The Test ID column in each of the tables provides a unique identifier for each test case. The test cases can be classified into two different types: functional and exception. A functional test type describes a normal use case where the action is designed to verify the successful invocation of a function call to accomplish certain tasks. An exception test type describes a test case were the action is designed to cause errors within specific components of the audit subsystem. The purpose of these tests is to verify that the audit subsystem is able to handle exceptions and exhibits expected behavior under such circumstances. The Action column gives a summary of the actions performed during the test, and preconditions for the test cases are described where applicable. The Expected Result column describes the expected behavior of the component for each test case.

Table 15. Function Test for audit_enabled

audit_enabled				
Test ID	**Test Type**	**Action**	**Expected Result**	**Pass / Fail**
F1-1	Functional	Call function when audit is not enabled	FALSE is returned	Pass
F1-2	Exception	Call function when audit is enabled	TRUE is returned	Pass

Table 16. Function Test for audit_write_INI_cv

audit_write_INI_cv				
Test ID	**Test Type**	**Action**	**Expected Result**	**Pass / Fail**
F2-1	Functional	Call function when audit is enabled. Provide valid input arguments.	AUD_NO_ERR is returned. Record is successfully written to the Audit Buffer.	Pass
F2-2	Exception	Call function when audit is disabled	AUD_ERR_DISABLED error code is returned	Pass
F2-3	Exception	Provide null pointer as arguments	AUD_ERR_INVALID_PARAM error code is returned	Pass
F2-4	Exception	Set the length of descriptive text to 0	AUD_ERR_INVALID_PARAM error code is returned	Pass

Table 17. Function Test for audit_write_INI_complete

audit_write_INI_complete				
Test ID	**Test Type**	**Action**	**Expected Result**	**Pass / Fail**
F3-1	Functional	Call function when audit is enabled.	AUD_NO_ERR is returned. Record is successfully written to the Audit Buffer.	Pass
F3-2	Exception	Call function when audit is disabled.	AUD_ERR_DISABLED error code is returned	Pass

Table 18. Function Test for audit_write_SYS_auditstart

audit_write_SYS_auditstart				
Test ID	**Test Type**	**Action**	**Expected Result**	**Pass / Fail**
F4-1	Functional	Call function when audit is enabled. Event modifier is set to AUD_MOD_START	AUD_NO_ERR is returned. Record is successfully written to the Audit Buffer.	Pass
F4-2	Functional	Call function when audit is enabled. Event modifier is set to AUD_MOD_SHUTDOWN	AUD_NO_ERR is returned. Record is successfully written to the Audit Buffer.	Pass
F4-3	Exception	Call function when audit is enabled. Provide invalid event modifier.	AUD_ERR_INVALID_PARAM error code is returned	Pass
F4-4	Exception	Call function when audit is disabled.	AUD_ERR_DISABLED error code is returned	Pass

Table 19. Function Test for audit_write_SYS_lpskstart

audit_write_SYS_lpskstart				
Test ID	**Test Type**	**Action**	**Expected Result**	**Pass / Fail**
F5-1	Functional	Call function when audit is enabled.	AUD_NO_ERR is returned. Record is successfully written to the Audit Buffer.	Pass
F5-2	Exception	Call function when audit is disabled.	AUD_ERR_DISABLED error code is returned	Pass

Table 20. Function Test for audit_write_SYS_sak

audit_write_SYS_sak				
Test ID	**Test Type**	**Action**	**Expected Result**	**Pass / Fail**
F6-1	Functional	Call function when audit is enabled. Provide valid input arguments.	AUD_NO_ERR is returned. Record is successfully written to the Audit Buffer.	Pass
F6-2	Exception	Call function when audit is disabled.	AUD_ERR_DISABLED error code is returned	Pass
F6-3	Exception	Provide null pointer as arguments	AUD_ERR_INVALID_PARAM error code is returned	Pass
F6-4	Exception	Provide an out of bound partition ID as argument	AUD_ERR_INVALID_PARAM error code is returned	Pass

Table 21. Function Test for audit_write_MEM_swapin

audit_write_MEM_swapin				
Test ID	**Test Type**	**Action**	**Expected Result**	**Pass / Fail**
F7-1	Functional	Call function when audit is enabled. Provide valid input arguments.	AUD_NO_ERR is returned. Record is successfully written to the Audit Buffer.	Pass
F7-2	Exception	Call function when audit is disabled.	AUD_ERR_DISABLED error code is returned	Pass
F7-3	Exception	Provide null pointer as arguments	AUD_ERR_INVALID_PARAM error code is returned	Pass
F7-4	Exception	Set dseg path length input argument to 0	AUD_ERR_INVALID_PARAM error code is returned	Pass

Table 22. Function Test for audit_write_MEM_msegcreate

audit_write_MEM_msegcreate				
Test ID	**Test Type**	**Action**	**Expected Result**	**Pass / Fail**
F8-1	Functional	Call function when audit is enabled. Provide valid input arguments.	AUD_NO_ERR is returned. Record is successfully written to the Audit Buffer.	Pass
F8-2	Exception	Call function when audit is disabled.	AUD_ERR_DISABLED error code is returned	Pass
F8-3	Exception	Provide null pointer as arguments	AUD_ERR_INVALID_PARAM error code is returned	Pass
F8-4	Exception	Provide an out of bound mseg ID as argument	AUD_ERR_INVALID_PARAM error code is returned	Pass

Table 23. Function Test for audit_write_SYN_ecawake

audit_write_SYN_ecawake				
Test ID	**Test Type**	**Action**	**Expected Result**	**Pass / Fail**
F9-1	Functional	Call function when audit is enabled. Provide valid input arguments.	AUD_NO_ERR is returned. Record is successfully written to the Audit Buffer.	Pass
F9-2	Exception	Call function when audit is disabled.	AUD_ERR_DISABLED error code is returned	Pass
F9-3	Exception	Provide null pointer as arguments	AUD_ERR_INVALID_PARAM error code is returned	Pass
F9-4	Exception	Provide an out of bound eventcount ID as argument	AUD_ERR_INVALID_PARAM error code is returned	Pass

Table 24. Function Test for audit_write_SYN_procawait

audit_write_SYN_procawait				
Test ID	**Test Type**	**Action**	**Expected Result**	**Pass / Fail**
F10-1	Functional	Call function when audit is enabled. Provide valid input arguments.	AUD_NO_ERR is returned. Record is successfully written to the Audit Buffer.	Pass
F10-2	Exception	Call function when audit is disabled.	AUD_ERR_DISABLED error code is returned	Pass
F10-3	Exception	Provide null pointer as arguments	AUD_ERR_INVALID_PARAM error code is returned	Pass
F10-4	Exception	Provide an out of bound eventcount ID as argument	AUD_ERR_INVALID_PARAM error code is returned	

Table 25. Function Test for audit_write_SYN_seqticket

audit_write_SYN_seqticket				
Test ID	**Test Type**	**Action**	**Expected Result**	**Pass / Fail**
F11-1	Functional	Call function when audit is enabled. Provide valid input arguments.	AUD_NO_ERR is returned. Record is successfully written to the Audit Buffer.	Pass
F11-2	Exception	Call function when audit is disabled.	AUD_ERR_DISABLED error code is returned	Pass
F11-3	Exception	Provide null pointer as arguments	AUD_ERR_INVALID_PARAM error code is returned	Pass
F11-4	Exception	Provide an out of bound sequencer ID as argument	AUD_ERR_INVALID_PARAM error code is returned	Pass

2. Testing of Exported Interfaces to Audit Retrieval

Tables 26 through 30 describe the testing of the functions exported by the audit subsystem to authorized subjects. A test application residing in PL3 was created to facilitate the testing. The API calls were invoked from the test application and the results were displayed on the screen for verification.

Table 26. Function Test for audit_read_next

audit_read_next				
Test ID	**Test Type**	**Action**	**Expected Result**	**Pass / Fail**
F12-1	Functional	Call function when audit is enabled. Request to read 1 record from the Audit Buffer.	AUD_NO_ERR is returned. Record is successfully read from the Audit Buffer. Number of records read is correctly returned via the output parameter.	Pass
F12-2	Functional	Call function when audit is enabled. Request to read multiple records from the Audit Buffer.	AUD_NO_ERR is returned. Records are successfully read from the Audit Buffer. Number of records read is correctly returned via the output parameter.	Pass
F12-3	Exception	Call function when audit is disabled.	AUD_ERR_DISABLED error code is returned	Pass
F12-4	Exception	Provide null pointer as argument	AUD_ERR_INVALID_PARAM error code is returned	Pass
F12-5	Exception	Provide a buffer size that is smaller than the record length.	AUD_ERR_SIZE_EXCEED error code is returned	Pass
F12-6	Exception	Request to read 1 record when the Audit Buffer is empty.	A value of 0 is returned for number of records read. AUD_ERR_BUF_EMPTY error code is also returned.	Pass
F12-7	Functional	Request to read more	All the records in the Audit	Pass

		records than is in the Audit Buffer. (E.g. Request to read 3 records when there is only 2 record in the Audit Buffer)	Buffer are read. Number of records read is correctly returned via the output parameter.	
F12-8	Functional	Request to read more than 1 record but the size of the buffer is not enough to receive all the records requested.	Records that will fit into the buffer are read. Number of records read is correctly returned via the output parameter.	

Table 27. Function Test for audit_read_buffer_size

audit_read_buffer_size				
Test ID	**Test Type**	**Action**	**Expected Result**	**Pass / Fail**
F13-1	Functional	Call function when audit is enabled. Provide valid argument to store size of Audit Buffer.	AUD_NO_ERR is returned. Size of Audit Buffer is correctly returned via the output parameter.	Pass
F13-2	Exception	Call function when audit is disabled.	AUD_ERR_DISABLED error code is returned	Pass
F13-3	Exception	Provide null pointer as argument	AUD_ERR_INVALID_PARAM error code is returned	Pass

Table 28. Function Test for audit_read_num_rec

audit_read_num_rec				
Test ID	**Test Type**	**Action**	**Expected Result**	**Pass / Fail**
F14-1	Functional	Call function when audit is enabled. Provide valid argument to store the number of records in the Audit Buffer.	AUD_NO_ERR is returned. Number of records in the Audit Buffer is correctly returned via the output parameter.	Pass
F14-2	Exception	Call function when audit is disabled.	AUD_ERR_DISABLED error code is returned	Pass
F14-3	Exception	Provide null pointer as argument	AUD_ERR_INVALID_PARAM error code is returned	Pass

Table 29. Function Test for audit_read_num_generated

audit_read_num_generated				
Test ID	**Test Type**	**Action**	**Expected Result**	**Pass / Fail**
F15-1	Functional	Call function when audit is enabled. Provide valid argument to store total number of records generated.	AUD_NO_ERR is returned. Total number of records generated is correctly returned via the output parameter.	Pass
F15-2	Exception	Call function when audit is disabled.	AUD_ERR_DISABLED error code is returned	Pass
F15-3	Exception	Provide null pointer as argument	AUD_ERR_INVALID_PARAM error code is returned	Pass

Table 30. Function Test for audit_read_num_overwritten

audit_read_num_overwritten				
Test ID	**Test Type**	**Action**	**Expected Result**	**Pass / Fail**
F16-1	Functional	Call function when audit is enabled. Provide valid argument to store total number of records already overwritten.	AUD_NO_ERR is returned. Total number of records overwritten is correctly returned via the output parameter.	Pass
F16-2	Exception	Call function when audit is disabled.	AUD_ERR_DISABLED error code is returned	Pass
F16-3	Exception	Provide null pointer as argument	AUD_ERR_INVALID_PARAM error code is returned	Pass

3. Testing of Audit Buffer

Table 31 describes additional testing performed on the Audit Buffer. The correct implementation of the Audit Buffer provides assurance that audit records will not be accidentally modified or deleted during read or write operations.

Table 31. Test for Audit Buffer

Test ID	Test Type	Action	Expected Result	Pass / Fail
AB-1	Functional	Add and retrieve a record to/from the Audit Buffer one at a time. Iterate this process until the read / write process "wraps around" the Audit Buffer a few times.	Audit records successfully written and read from the Audit buffer.	Pass
AB-2	Exception	Add an audit record to the Audit Buffer when it is full, such that the newest record is of a different type than the record to be overwritten. (Audit Buffer is configured to overwrite old records.)	The oldest audit record is overwritten. The number of records overwritten and the number of records in the Audit Buffer is updated. Retrieve all the audit records and verify that that last record returned is the one that overwrote the oldest record.	Pass

The developmental testing proved to be very useful as a number of bugs were detected. Bugs related to the Audit Buffer were especially difficult to detect because they occur intermittently and are difficult to replicate. To make matters worse, the debugging process is very time consuming as every time a change is made, the files must be transferred from the development virtual machine to the test virtual machine and the test virtual machine must then be rebooted.

Nevertheless, Developmental testing has helped to provide a systematic way to isolate problems. All the bugs that were found were successfully corrected. The test was re-run successfully and no additional problem was found.

B. ACCEPTANCE TESTING

The purpose of the acceptance testing is to ensure the proper functioning of the audit subsystem to support the audit requirements of LPSK. The LPSK kernel source code was modified to generate various audit events. It was compiled together with the audit subsystem modules. The configuration vector was configured to create eventcounts, sequencers, msegs and dsegs for the purpose of audit record generation testing. After successful initialization, the LPSK modules start to invoke function calls to the audit subsystems to record the audit events.

A test application was also created to read the audit records from the Audit Buffer. The test application provides the user with a menu interface to select different types of requests to the audit subsystem. A user can use the menu to retrieve audit records from the Audit Buffer. Because the LPSK does not support a secondary storage device driver in the current prototype implementation, the test application creates another buffer to simulate a secondary storage device. Retrieved audit records are stored in this buffer. The test application can also read and display the audit records on the console. This allows manual inspection of the audit records to verify that audit event information is correctly captured in the record.

Table 32 describes the acceptance tests performed when audit is enabled. The actions are designed to systematically trigger the audit events implemented in this study.

Table 32. Acceptance Tests when Audit is Enabled

Test ID	Action	Expected Result	Pass / Fail
A-1	Boot up LPSK. Start the audit test application. Request for the number of audit records in the Audit Buffer.	Audit subsystem correctly returns the number of records generated	Pass
A-2	After doing A-1, request to read an audit record from the Audit Buffer, followed by a request for the number of audit records. Repeat until Audit Buffer is empty.	Audit records are successfully read from the Audit Buffer. The number of audit records in the Audit Buffer is decremented by 1 after each retrieval.	Pass
A-3	Reboot the LPSK and go to the audit testing partition. Display and verify the audit records.	Audit records for the following events are successfully returned. • start of LPSK audit subsystem • identifying information of the configuration vector • Swap in of a dseg defined in the configuration vector • Creation of a mseg defined in the configuration vector • Successful completion of LPSK initialization • Successful startup of LPSK runtime.	Pass

		Correct timestamp is attached to each record. CRC32 checksum is verified for each record.	
A-4	Verify that the audit buffer is empty, then invoke a SAK and then return to the audit testing partition. Read the audit record.	Audit record is returned for the detection of SAK	Pass
A-5	Verify that the audit buffer is empty, then request the ticket of a sequencer. (Test application has read and write permission to the sequencer)	Audit record (for success) is returned for this event.	Pass
A-6	Verify that the audit buffer is empty, then request an await on an eventcount. (Test application has read and write permission to the eventcount)	Audit record (for success) is returned for this event.	Pass
A-7	Verify that the audit buffer is empty, then advance the eventcount mentioned in A-6. (Test application has read and write permission to the eventcount)	Audit record (for success) is returned when the process wake from an await on eventcount.	Pass
A-8	Verify that the audit buffer is empty, then request the ticket of a sequencer. (Test application does not have	Audit record (for failure) is returned for this event.	Pass

	read and write permission to the sequencer)		
A-9	Verify that the audit buffer is empty, then request an await on an eventcount. (Test application does not have read and write permission to the eventcount)	Audit record (for failure) is returned for this event.	Pass
A-10	Keep invoking the SAK to generate enough records to fill up the Audit Buffer.	Audit records are generated. Old records are overwritten by new ones when the Audit Buffer is full. Number of audit records being overwritten is updated.	Pass

Table 33 describes the acceptance tests performed when audit is disabled. The purpose of this test is to ensure that LPSK continues to function properly when audit is disabled.

Table 33. Acceptance Tests when Audit is Disabled

Test ID	**Action**	**Expected Result**	**Pass / Fail**
B-1	Boot up LPSK. Start the audit test application. Request for the number of audit records in the buffer.	No audit record generated.	Pass
B-2	Request to read an audit record from the Audit Buffer.	No audit record in the Audit Buffer.	Pass
B-3	Perform the tasks described in A-4 to A-10 in Table 31	No audit record is returned.	Pass

The acceptance tests were successful. No bugs were found during acceptance testing because the functional and exception tests appear to have identified all the bugs. Detailed test procedures and results are provided in the Appendix.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSION

This study explored the best way to design and implement an audit subsystem for the LPSK. The first step was to gather the audit requirements that were sprinkled throughout the LPSK functional specifications and SKPP.

From these requirements, a list of auditable events was created. It was determined that the information required to describe and record each event differs a lot. Some events require more attributes to describe them than others. This means that the size of the audit records could vary. Due to the fact that managing varying length records adds considerable amount of complexity to the implementation, the idea of allocating a maximum fixed size for each record was initially explored. In general, such an approach would result in inefficient use of limited memory space. A varying length token based record format was found to be the better solution for LPSK audit records. The highly structured nature of the tokens helps to relieve the difficulties in managing varying length records.

Audit interfaces were defined to allow LPSK modules to send audit information to the audit subsystem and to allow an authorized application to retrieve audit records. The design is based on the assumption that only one authorized application retrieves records from the audit subsystem. No use case has been identified so far that would involve having multiple applications concurrently reading records from the Audit Buffer.

After the audit interfaces were defined, an LPSK audit module was designed to manage the audit buffers and other audit metadata. Because of the specified nature of the audit subsystem, it was determined that the best abstract data structure for managing the records was a circular buffer, which would allow old records to be read from one side of the buffer, and allow new records to be added to the other side of the buffer.

A prototype audit subsystem was developed to test out the design. Development and testing was time consuming due to the fact that every run of the code involves transferring compiled binaries from the development virtual machine to the testing virtual machine. When the LPSK is initializing, there is not much feedback on the status of the

audit subsystem to know whether things are going right. A debugger, such as the VMware Vprobes [19], would have been helpful in troubleshooting the code at the kernel level. However, Vprobes requires developers to write scripts to collect the data they want to investigate. This creates a learning curve for developers who are unfamiliar with Vprobes scripts. Furthermore, the lack of a LPSK disk device driver means that currently it is not possible to write debug logs to a secondary storage device. The workaround was to strategically place function calls to print debug messages to the screen and pause the initialization to be able to see the messages before they are overwritten by other messages.

Testing was conducted and problems were found due to incorrect implementation of the operations to read and write records to the audit buffer. The bugs were corrected and subsequent testing was completed successfully.

A. RELATED WORK

This section introduces related work on separation kernel audit subsystems. Green Hills' INTEGRITY-178 Operating Systems [20] was the first separation kernel to be certified compliant with the SKPP. Its audit subsystem bears close resemblance to the one implemented in this study due to the fact that both INTEGRITY-178 and LPSK draw their audit requirements from the SKPP.

The INTEGRITY-178's audit event log is stored using a circular buffer in kernel memory. Once read, the audit record is removed from the circular buffer. The oldest record will be overwritten when the buffer fills up. This implementation is very similar to the LPSK's implementation. The main difference is that the INTEGRITY-178 abstracts the circular buffer as an I/O device object. Accesses to the I/O devices are configured using static configuration files. No information regarding the format of the audit records for INTEGRITY-178 is available.

The LynxSecure Embedded Hypervisor by LynuxWorks [21] and VxWorks by Wind Rivers Systems [22] are another two separation kernels undergoing certification to be compliant with SKPP. However, there was no information available about their implementations of the audit subsystems.

B. FUTURE WORK

This section presents some recommendations for future work.

1. Abstraction of Audit Subsystem as a Device

Before finding the related work from Green Hills, it was already suggested that future work could look into the possibility of abstracting the audit interface like a device. The interface to the audit subsystem is very similar to a device interface, especially to an asynchronous read-only device like a keyboard. Both are using internal buffers to store data while exporting a set of kernel APIs to allow external applications to obtain the buffered information. The following are the potential benefits to this abstraction:

- Access control to the audit API can be controlled in the same way the devices are controlled.
- The audit metadata can be made available through the device CONTROL interface.
- The number of kernel APIs is reduced, thus reducing kernel complexity.

2. Audit Review

This study has focused on the design of an audit subsystem to generate and collect audit events within the LPSK, and to provide an interface to allow authorized subjects to extract audit records from the kernel and store them on secondary storage for future review. While a token-based audit record format is suitable within the LPSK, it may not be the ideal format for human review. The overall audit system would not be complete without providing an effective way for the administrator to review the audit records. A detailed study needs to be done to look at the best way to store, process and present the records to the administrator. This would also include exploring how records will be put into, and retrieved from, a secondary storage device.

3. Performance Study

In this study, it was decided that as long as audit is enabled, the LPSK modules will always send event information to the audit subsystem when a potential auditable

event has occurred. The audit subsystem checks the audit policy to decide whether the event should be recorded. While this approach is simple, a potential performance penalty may be incurred if a large portion of the events sent to the audit subsystem do not need to be audited. A study to assess the performance impact of such an approach is needed in order to determine the best way to perform the audit policy checking.

4. Implementation of Unfinished Work

Due to the fact that the LPSK prototype is currently incomplete, several of the audit subsystem features were not implemented in this work. The following is a list of work that needs to be done when the prototype is more fully developed:

- Initialize the audit subsystem based on the configuration read from the configuration vector. This includes behavior of the circular buffer when it is full and advanced filtering rules for selective auditing of events based on attributes, which include subject identity, resource identity, event type, and success or failure of particular events.
- Modify the header token to accept timestamps with finer granularity. The current LPSK prototype uses epoch time that can only measure to a granularity of one second, But it is expected that a future version of the LPSK will provide more granularity.
- Implement auditing of all 46 auditable events identified in this study.

C. CONCLUSION

The LPSK provides a high assurance platform that could potentially be used to protect sensitive data in both public and private sectors. In order to ensure that accountability policies are being enforced correctly, and that no one is abusing their privileged access, there is a need for a mechanism to allow administrators to regularly review events. The audit subsystem prototype developed in this study has demonstrated a working mechanism to efficiently collect audit records and transfer them to an authorized application, which will then store or process these records for viewing by an

administrator. Even though the audit prototype is not yet a complete implementation, it provides a good environment to study the various features of the audit design.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX

This appendix describes the test procedures for the test plan used in Chapter V.

A. DEVELOPMENTAL TESTING

Conditional test code was added to the LPSK modules to help to test the functions exported by the audit subsystem. The test code invokes the audit functions using different input arguments and sometimes under different conditions, prints debug messages and displays values to the screen. To enable the test code, do the following:

1. Find the line “wcc386 kernel_ini2.c $(INC) $(CC_OPTS)” in the Makefile and append the debug option “-DDEBUG_AUDIT_DEV” to the end of the line.
2. Uncomment the line “#define AUDIT_ENABLED 1” and comment the line “#define AUDIT_ENABLED 0” in the lpsk_audit.h file to enable audit.
3. Compile the LPSK code with the new Makefile.
4. Copy the compiled binary to the test VM and power it on.

After completing the above, the test code performs tests on various function calls and prints the results to the screen. Table 34 describes the expected results for the tests described in Chapter V when the audit is enabled. The function return code of AUD_NO_ERR is defined as 0 and AUD_ERR_INVALID_PARAM is defined as 1.

Table 34. Testing Results of Interfaces to Kernel Modules when Audit is Enabled

Test ID	Expected Results Summary
F1-1	F1-1: Return code = 1
F2-1	F2-1: Return code = 0
F2-3	F2-3: Return code = 1
F2-4	F2-4: Return code = 1

F3-1	F3-1: Return code = 0
F4-1	F4-1: Return code = 0
F4-2	F4-2: Return code = 0
F4-3	F4-3: Return code = 1
F5-1	F5-1: Return code = 0
F6-1	F6-1: Return code = 0
F6-3	F6-3: Return code = 1
F6-4	F6-4: Return code = 1
F7-1	F7-1: Return code = 0
F7-3	F7-3: Return code = 1
F7-4	F7-4: Return code = 1
F8-1	F8-1: Return code = 0
F8-3	F8-3: Return code = 1
F8-4	F8-4: Return code = 1
F9-1	F9-1: Return code = 0
F9-3	F9-3: Return code = 1
F9-4	F9-4: Return code = 1
F10-1	F10-1: Return code = 0
F10-3	F10-3: Return code = 1
F10-4	F10-4: Return code = 1
F11-1	F11-1: Return code = 0
F11-3	F11-3: Return code = 1
F11-4	F11-4: Return code = 1

Disable audit by doing the following:

1. Commenting the line “#define AUDIT_ENABLED 1”
2. Uncommenting the line “#define AUDIT_ENABLED 0” in the lpsk_audit.h file.
3. Compile the LPSK code.
4. Copy the binary to the test VM and power it on.

Table 35 describes the expected results for the tests described in Chapter V when the audit is disabled. The function return code of AUD_ERR_DISABLED is defined as 2.

Table 35. Testing Results of Interfaces to Kernel Modules when Audit is Disabled

Test ID	**Expected Results Summary**
F1-2	F1-1: Return code = 0
F2-2	F2-1: Return code = 2
F3-2	F3-2: Return code = 2
F4-4	F4-4: Return code = 2
F5-2	F5-2: Return code = 2
F6-2	F6-2: Return code = 2
F7-2	F7-2: Return code = 2
F8-2	F8-2: Return code = 2
F8-3	F8-3: Return code = 2
F8-4	F8-4: Return code = 2
F9-2	F9-2: Return code = 2
F10-2	F10-2: Return code = 2

F11-2	F11-2: Return code = 2

To test the functions exported by the audit subsystem to authorized subjects, take the following steps:

1. Set the size of the Audit Buffer to 1024 bytes via the AUDIT_BUFF_SIZE constant defined in lpsk_audit.h. Enable audit by setting the AUDIT_ENABLED constant in lpsk_audit.h to TRUE. Compile the code with "-DDEBUG_AUDIT_DEV" to generate audit events in the Audit Buffer for testing.
2. Boot the Test Virtual Machine and login to Trusted Path Application
3. Select "F – Change Partition Focus" option from the menu
4. Select "1 – Audit Application" option from the menu
5. From the main menu of the audit test application perform the steps shown in the "Menu Selection / Action" column of Table 36. The "Expected Results Summary" column shows the expected message displayed by the test application after each step has been performed.

Table 36. Testing Results of Interfaces to Audit Retrieval when Audit is Enabled

Test ID	Menu Selection / Action	Expected Results Summary
F13-1	Select "3 – Query size of audit buffer"	Size of audit buffer = 1024 Return code = 0 (Remarks: return code 0 = AUD_NO_ERR)
F14 -1	Select "4 – Query number of records in the Audit Buffer"	Number of records in the buffer = 19 Return code = 0

F12-1	Select “1 – Retrieve records”. Enter “1” when asked to enter the number of records to retrieve.	Number of records read = 1 Return code = 0
F12-2	Select “1 – Retrieve records”. Enter “3” when asked to enter the number of records to retrieve.	Number of records read = 3 Return code = 0
F12-7	Retrieve 13 more records and leave 2 in the Audit Buffer. Select “1 – Retrieve records”. Enter “3” when asked to enter the number of records to retrieve.	Number of records read = 2 Return code = 0
F12-8	Press “Alt-Esc” to generate a SAK, and then change the partition focus back to the audit application. Repeat the above mentioned action 5 times to generate 6 SAK events in the Audit Buffer. Select “1 – Retrieve records”. Enter “6” when asked to enter the number of records to retrieve.	Number of records read = 5 Return code = 0 (Remarks: the size of the buffer provided by the test application is 128 bytes, which is only enough to hold 5 SAK events of 22 bytes each)
F12-4 F12-5 F13-3 F14-3 F15-3 F16-3	Select “7 – Exception Testing”	F12-4: Return code = 1 F12-5: Return code = 5 F13-3: Return code = 1 F14-3: Return code = 1 F15-3: Return code = 1 F16-3: Return code = 1 (Remarks: Return code 1 = AUD_ERR_INVALID_PARAM;

		Return code 5 = AUD_ERR_SIZE_EXCEED)
F15-1	Select "6 – Query total number of records generated"	Number of records generated = 25 Return code = 0
F16-1	Select "5 – Query number of overwritten records"	Number of records overwritten = 0 Return code = 0

To perform the tests when audit is disabled, do the following:

1. Disable audit by setting the AUDIT_ENABLED constant in lpsk_audit.h to FALSE.
2. Recompile the code and boot up the Test Virtual Machine.
3. Navigate to the audit test application main menu.
4. Perform the steps shown in the "Menu Selection / Action" column of Table 37.

Table 37. Testing Results of Interfaces to Audit Retrieval when Audit is Disabled

Test ID	**Menu Selection / Action**	**Expected Results Summary**
F12-3	Select "1 – Retrieve records". Enter "1" when asked to enter the number of records to retrieve.	Return code = 2 (Remarks: return code 2 = AUD_ERR_DISABLED)
F13-2	Select "3 – Query size of audit buffer"	Return code = 2
F14-2	Select "4 – Query number of records in the Audit Buffer"	Return code = 2
F15-2	Select "6 – Query total number of records	Return code = 2

	generated”	
F16-2	Select “5 – Query number of overwritten records”	Return code = 2

B. ACCEPTANCE TESTING

The acceptance tests verified that audit records are generated for a set of predefined events. The configuration vector needs to be configured to ensure that mseg, dseg, sequencer and eventcount events are successfully generated during the acceptance tests. Perform the following steps:

1. Insert the following lines in the LPSK initialization database file all_apps.pl0.
    a. EVENTCOUNT = {"Event 0", 3, RW, RW, NA, RW, RW };
    b. SEQUENCER = {"Seq 0", 3, RW, RW, NA, RW, RW };
2. Insert the following lines in the PL3 initialization database file all_apps.pl3
    a. MSEG[0] = { 40000, 1, RW, RW, NA, RO, RO };
3. Create the configuration vector using the command line “vector -0 all_aps.pl0 -1 five_part.pl1 -2 five_part.pl2 -3 all_apps.pl3 –o all_apps”
4. Re-enable audit by setting the AUDIT_ENABLED constant in lpsk_audit.h to TRUE.
5. Compile the code without the debug option.
6. Boot the Test Virtual Machine and navigate to the audit test application main menu. Perform the steps shown in the “Menu Selection / Action” column of Table 38.

Table 38. Results of Acceptance Testing (Successful Events)

Test ID	Menu Selection / Action	Expected Results Summary
A-1	Select "4 – Query number of records in the Audit Buffer"	Number of records in the buffer = 8 Return code = 0
A-2	Select "1 – Retrieve records". Enter "1" when asked to enter the number of records to retrieve.	Number of records read = 1 Return code = 0
	Select "4 – Query number of records in the Audit Buffer"	Number of records in the buffer = 7 Return code = 0
	Repeat A-2 until buffer is empty	The number of records in the Audit Buffer is decremented by 1 after each retrieval.
A-3	Select "2 – View records" to display and manually inspect the audit records	Content of the audit records for the following events • start of LPSK audit subsystem • identifying information of the configuration vector • Swap in of 2 dsegs defined in the configuration vector

		• Creation of 2 msegs defined in the configuration vector • Successful completion of LPSK initialization • Successful startup of LPSK runtime.
A-4	Select “4 – Query number of records in the Audit Buffer”	Number of records in the buffer = 0 Return code = 0
	Press “Alt-Esc” to generate a SAK, and then change the partition focus back to the audit application. Select “1 – Retrieve records”. Enter “1” when asked to enter the number of records to retrieve.	Number of records read = 1 Return code = 0
	Select “2 – View records” to display and manually inspect the audit records	Content of the audit record for the SAK event
A-5	Select “4 – Query number of records in the Audit Buffer”	Number of records in the buffer = 0 Return code = 0
	• Press “Alt-Esc” to generate a SAK, and then change the partition focus to “3 – Test Application”. • Select “A – Test eventcounts and	Number of records read = 4 Return code = 0 (Remarks: The test program will generate 2 sequencer

<table>
<tr><td></td><td>sequencers” from the Test Menu.
Enter ‘Q’ to quit the “read and advance eventcount” test
Enter ‘0’ when prompted to enter the sequencer to ticket
Press “Alt-Esc” to generate a SAK, and then change the partition focus back to the audit application
Select “1 – Retrieve records”. Enter “4” when asked to enter the number of records to retrieve.</td><td>events. The other 2 audit records are generated by the SAK events)</td></tr>
<tr><td></td><td>Select “2 – View records” to display and manually inspect the audit records</td><td>Content of the audit record for the sequencer event</td></tr>
<tr><td>A-6</td><td>Select “4 – Query number of records in the Audit Buffer”</td><td>Number of records in the buffer = 0

Return code = 0</td></tr>
<tr><td></td><td>Press “Alt-Esc” to generate a SAK, and then change the partition focus to “3 – Test Application”.
Enter ‘Q’ to quit the sequencer test
Enter ‘Y’ to continue with the “Await on Eventcount” test.
Press “Alt-Esc” to generate a SAK, and then change the partition focus back to the audit application
Select “1 – Retrieve records”. Enter “3”</td><td>Number of records read = 3

Return code = 0

(Remarks: 2 of the audit records are generated by the SAK events)</td></tr>
</table>

	when asked to enter the number of records to retrieve.	
	Select “2 – View records” to display and manually inspect the audit records	Content of the audit record for the await on eventcount event
A-7	Select “4 – Query number of records in the Audit Buffer”	Number of records in the buffer = 0 Return code = 0
	• Press “Alt-Esc” to generate a SAK, and then change the partition focus to “4 – Test Support”. • Enter ‘0’ when prompted for the eventcount to advance. • Press “Alt-Esc” to generate a SAK, and then change the partition focus back to the audit application • Select “1 – Retrieve records”. Enter “3” when asked to enter the number of records to retrieve.	Number of records read = 3 Return code = 0 (Remarks: 2 of the audit records are generated by the SAK events)
	Select “2 – View records” to display and manually inspect the audit records	Content of the audit record for the process woke up event

Shut down the Test Virtual Machine. To perform tests to verify that audit records are generated for failed events, reconfigure the configuration vector by performing the following steps:

1. Insert the following lines in all_apps.pl0
    a. EVENTCOUNT = {"Event 0", 3, RW, RW, NA, NA, NA };
    b. SEQUENCER = {"Seq 0", 3, RW, RW, NA, NA, NA };

2. Create the configuration vector using the command line “vector -0 all_aps.pl0 -1 five_part.pl1 -2 five_part.pl2 -3 all_apps.pl3 –o all_apps”
3. Compile the code without the debug option.
4. Boot the Test Virtual Machine and navigate to the audit test application main menu. Perform the steps shown in the “Menu Selection / Action” column of Table 39.

Table 39. Results of Acceptance Testing (Failed Events)

Test ID	Menu Selection / Action	Expected Results Summary
A-8	Select “1 – Retrieve records”. Enter “6” when asked to enter the number of records to retrieve. The buffer provided by the test application is not big enough to hold all 8 records. Repeat this step several times until number of records read becomes 0.	All the records are read.
	Select “2 – View records” 8 times to display all the audit records. Select “4 – Query number of records in the Audit Buffer”	Number of records in the buffer = 0 Return code = 0
	• Press “Alt-Esc” to generate a SAK, and then change the partition focus to “3 – Test Application”. • Select “A – Test eventcounts and sequencers” from the Test Menu. • Enter ‘Q’ to quit the “read and advance	Number of records read = 3 Return code = 0 (Remarks: 2 of the audit records are generated by the SAK events)

	eventcount" test • Enter '0' when prompted to enter the sequencer to ticket • Press "Alt-Esc" to generate a SAK, and then change the partition focus back to the audit application • Select "1 – Retrieve records". Enter "3" when asked to enter the number of records to retrieve.	
	Select "2 – View records" to display and manually inspect the audit records	Content of the audit record for the sequencer event (Event modifier = 0x1)
A-9	Select "4 – Query number of records in the Audit Buffer"	Number of records in the buffer = 0 Return code = 0
	• Press "Alt-Esc" to generate a SAK, and then change the partition focus to "3 – Test Application". • Enter 'Q' to quit the sequencer test • Enter 'Y' to continue with the "Await on Eventcount" test. • Press "Alt-Esc" to generate a SAK, and then change the partition focus back to the audit application • Select "1 – Retrieve records". Enter "3" when asked to enter the number of	Number of records read = 3 Return code = 0 (Remarks: 2 of the audit records are generated by the SAK events)

	records to retrieve.	
	Select "2 – View records" to display and manually inspect the audit records	Content of the audit record for the await on eventcount event (Event modifier = 0x1)

Shut down the Test Virtual Machine. To perform testing of the Audit Buffer, modify the size of the Audit Buffer by performing the following steps:

1. Modify the value of AUDIT_BUFF_SIZE constant in lpsk_audit.h to 512.
2. Recompile the code without the debug option.
3. Boot the Test Virtual Machine and navigate to the audit test application main menu. Perform the steps shown in the "Menu Selection / Action" column in Table 40.

Table 40. Results of Acceptance Testing (Audit Buffer)

Test ID	Menu Selection / Action	Expected Results Summary
A-10	Select "4 – Query number of records in the Audit Buffer"	Number of records in the buffer = 8 Return code = 0
	• Press "Alt-Esc" to generate a SAK. Repeat 10 times to generate 11 SAK audit records. • Change the partition focus back to the audit application. • Select "4 – Query number of records in the Audit Buffer"	Number of records in the buffer = 18 Return code = 0

	Select "5 – Query number of overwritten records"	Number of records overwritten = 1 Return code = 0

Shut down the Test Virtual Machine and disable the audit by performing the following steps:

1. Modify the value of AUDIT_ENABLED constant in lpsk_audit.h to FALSE.
2. Recompile the code without the debug option.
3. Boot up the Test Virtual Machine and navigate to the audit test application main menu. Perform the steps shown in the "Menu Selection / Action" column in Table 41.

Table 41. Results of Acceptance Testing when Audit is Disabled

Test ID	Menu Selection / Action	Expected Results Summary
B-1	Select "4 – Query number of records in the Audit Buffer"	Number of records in the buffer = 0 Return code = 2
	Select "1 – Retrieve records". Enter "1" when asked to enter the number of records to retrieve.	Number of records in the buffer = 0 Return code = 2
	Perform the tasks described in A-4 to A9. Select "4 – Query number of records in the Audit Buffer"	Number of records in the buffer = 0 Return code = 2

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] C. E. Irvine, T. E, Levin, P. C. Clark and T. D. Nguyen, "A Security Architecture for Transient Trust," in *Proceedings of the 2nd ACM Workshop on Computer Security Architectures*, Alexandria, VA, 2008, pp. 1–8.

[2] C. E. Irvine, T. E. Levin, T. D. Nguyen, and G. W. Dinolt, "The Trusted Computing Exemplar Project," in *Proceedings of the 5th IEEE Systems Man and Cybernautics Information Assurance Workshop*, West Point, NY, June 2004, pp. 109–115.

[3] IAD (Information Assurance Directorate), "U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness," National Information Assurance Partnership, version 1.03 ed., 29 June 2007.

[4] "Recommended Security Controls for Federal Information Systems," National Institute of Standards and Technology, 2009.

[5] P. C. Clark, D. J. Shifflett, C. E. Irvine, T. D. Nguyen, and T. E. Levin, "Trusted Computing Exemplar Least Privilege Separation Kernel Product Functional Specification," Naval Postgraduate School Center for Information Systems Security Studies and Research, 2010.

[6] J. M. Rushby, "Design and Verification of Secure Systems," in *ACM SIGOPS Operating Systems Review*, 15, 5, pp. 12–21, 1981.

[7] J. H. Saltzer and M. D. Schroeder, "The Protection of Information in Operating Systems," in Proceedings of IEEE, 63, 9, pp. 1278–1308, 1975.

[8] "Common Criteria documentation," July 2009, http://www.commoncriteriaportal.org/cc.

[9] "Minimum Security Requirements for Federal Information and Information Systems," National Institute of Standards and Technology, FIPS-200, March 2006.

[10] "A Guide to Understanding Audits in Trusted System," National Computer Security Center, NCSC-TG-001, version 2, 1 June 1988.

[11] S. Harris, *CISSP All in One Exam Guide* (4th ed.). New York:McGraw-Hill, 2007.

[12] K. Kent, M. Souppaya, "Guide to Computer Security Log Management," National Institute of Standards and Technology, September 2006.

[13] "Extensible Markup Language (XML) 1.0 (Fifth Edition)," 26 November 2008, http://www.w3.org/TR/2008/REC-xml-20081126.

[14] "RFC3164 - The BSD Syslog Protocol," August 2001, http://www.faqs.org/rfcs/rfc3164.html.

[15] "Open Source Sendmail," http://www.sendmail.org.

[16] "Event Log File Format," 20 May 2010, http://msdn.microsoft.com/en-us/library/bb309026(v=VS.85).aspx.

[17] "Trusted Solaris Audit Administration," December 2000, http://dlc.sun.com/pdf/805-8121/805-8121.pdf.

[18] "IEEE Std 1003.1," 2004, http://www.unix.org/version3/ieee_std.html.

[19] "Vprobes Programming Reference," 2008, http://www.vmware.com/products/beta/ws/vprobes_reference.pdf.

[20] "Green Hills Software Integrity 178-B Separation Kernel Security Target," 30 May 2008, http://www.niap-ccevs.org/st/st_vid10119-st.pdf.

[21] "LynxSecure Embedded Hypervisor and Separation Kernel," 2010, http://www.lynuxworks.com/virtualization/lynxsecure-hypervisor.pdf.

[22] "Wind River VxWorks," http://www.windriver.com/products/vxworks.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, VA

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, CA

3. Professor Tat Soon Yeo
   Temasek Defence Systems Institute (TDSI)
   National University of Singapore

4. Ms Tan Lai Poh
   Temasek Defence Systems Institute (TDSI)
   National University of Singapore

5. Kris Britton
   National Security Agency
   Fort Meade, MD

6. John Campbell
   National Security Agency
   Fort Meade, MD

7. Deborah Cooper
   DC Associates, LLC
   Reston, VA

8. Grace Crowder
   NSA
   Fort Meade, MD

9. Louise Davidson
   National Geospatial Agency
   Bethesda, MD

10. Vincent J. DiMaria
    National Security Agency
    Fort Meade, MD

11. Rob Dobry
NSA
Fort Meade, MD

12. Jennifer Guild
SPAWAR
Charleston, SC

13. CDR Scott Heller
SPAWAR
Charleston, SC

14. Dr. Steven King
ODUSD
Washington, DC

15. Steve LaFountain
NSA
Fort Meade, MD

16. Dr. Greg Larson
IDA
Alexandria, VA

17. Dr. Carl Landwehr
National Science Foundation
Arlington, VA

18. Dr. John Monastra
Aerospace Corporation
Chantilly, VA

19. John Mildner
SPAWAR
Charleston, SC

20. Dr. Victor Piotrowski
National Science Foundation
Arlington Virginia

21. Jim Roberts
Central Intelligence Agency
Reston, VA

22. Ed Schneider
IDA
Alexandria, VA

23. Mark Schneider
NSA
Fort Meade, MD

24. Keith Schwalm
Good Harbor Consulting, LLC
Washington, DC

25. Ken Shotting
NSA
Fort Meade, MD

26. Dr. Ralph Wachter
ONR
Arlington, VA

27. Dr. Cynthia E. Irvine
Naval Postgraduate School
Monterey, CA

28. Paul C. Clark
Naval Postgraduate School
Monterey, CA

29. Boon Pin Toh
Naval Postgraduate School
Monterey, CA